# FingerTrees

Benedikt Nordhoff Stefan Körner Peter Lammich

April 19, 2020

**Abstract**

We implement and prove correct 2-3 finger trees. Finger trees are a general purpose data structure, that can be used to efficiently implement other data structures, such as priority queues. Intuitively, a finger tree is an annotated sequence, where the annotations are elements of a monoid. Apart from operations to access the ends of the sequence, the main operation is to split the sequence at the point where a *monotone predicate* over the sum of the left part of the sequence becomes true for the first time. The implementation follows the paper of Hinze and Paterson[1]. The code generator can be used to get efficient, verified code.

## Contents

# 1 2-3 Finger Trees

**theory** *FingerTree*
**imports** *Main*
**begin**

We implement and prove correct 2-3 finger trees as described by Ralf Hinze and Ross Paterson[1].

This theory is organized as follows: Section 1.1 contains the finger-tree datatype, its invariant and its abstraction function to lists. The Section 1.2 contains the operations on finger trees and their correctness lemmas. Section 1.3 contains a finger tree datatype with implicit invariant, and, finally, Section 1.4 contains a documentation of the implemented operations.

**Technical Issues** As Isabelle lacks proper support of namespaces, we try to simulate namespaces by locales.

The problem is, that we define lots of internal functions that should not be exposed to the user at all. Moreover, we define some functions with names equal to names from Isabelle's standard library. These names make perfect sense in the context of FingerTrees, however, they shall not be exposed to anyone using this theory indirectly, hiding the standard library names there.

Our approach puts all functions and lemmas inside the locale *FingerTree_loc*, and then interprets this locale with the prefix *FingerTree*. This makes all definitions visible outside the locale, with qualified names. Inside the locale, however, one can use unqualified names.

## 1.1 Datatype definition

**locale** *FingerTreeStruc-loc*

Nodes: Non empty 2-3 trees, with all elements stored within the leafs plus a cached annotation

**datatype** $('e,'a)$ *Node* = *Tip* $'e$ $'a$ |
*Node2* $'a$ $('e,'a)$ *Node* $('e,'a)$ *Node* |
*Node3* $'a$ $('e,'a)$ *Node* $('e,'a)$ *Node* $('e,'a)$ *Node*

Digit: one to four ordered Nodes

**datatype** $('e,'a)$ *Digit* = *One* $('e,'a)$ *Node* |
*Two* $('e,'a)$ *Node* $('e,'a)$ *Node* |
*Three* $('e,'a)$ *Node* $('e,'a)$ *Node* $('e,'a)$ *Node* |
*Four* $('e,'a)$ *Node* $('e,'a)$ *Node* $('e,'a)$ *Node* $('e,'a)$ *Node*

FingerTreeStruc: The empty tree, a single node or some nodes and a deeper tree

**datatype** $('e, 'a)$ *FingerTreeStruc* =
*Empty* |
*Single* $('e,'a)$ *Node* |
*Deep* $'a$ $('e,'a)$ *Digit* $('e,'a)$ *FingerTreeStruc* $('e,'a)$ *Digit*

### 1.1.1 Invariant

**context** *FingerTreeStruc-loc*
**begin**

### Auxiliary functions

Readout the cached annotation of a node

**primrec** *gmn* :: $('e,'a$::*monoid-add*$)$ *Node* $\Rightarrow$ $'a$ **where**
*gmn* (*Tip e a*) = *a* |
*gmn* (*Node2 a - -*) = *a* |
*gmn* (*Node3 a - - -*) = *a*

The annotation of a digit is computed on the fly

**primrec** *gmd* :: $('e,'a$::*monoid-add*$)$ *Digit* $\Rightarrow$ $'a$ **where**
*gmd* (*One a*) = *gmn a* |
*gmd* (*Two a b*) = (*gmn a*) + (*gmn b*)|
*gmd* (*Three a b c*) = (*gmn a*) + (*gmn b*) + (*gmn c*)|
*gmd* (*Four a b c d*) = (*gmn a*) + (*gmn b*) + (*gmn c*) + (*gmn d*)

Readout the cached annotation of a finger tree

**primrec** *gmft* :: $('e,'a$::*monoid-add*$)$ *FingerTreeStruc* $\Rightarrow$ $'a$ **where**
*gmft Empty* = *0* |
*gmft* (*Single nd*) = *gmn nd* |
*gmft* (*Deep a - - -*) = *a*

Depth and cached annotations have to be correct

**fun** *is-leveln-node* :: *nat* $\Rightarrow$ $('e,'a)$ *Node* $\Rightarrow$ *bool* **where**
*is-leveln-node 0* (*Tip - -*) $\longleftrightarrow$ *True* |
*is-leveln-node* (*Suc n*) (*Node2 - n1 n2*) $\longleftrightarrow$
*is-leveln-node n n1* $\wedge$ *is-leveln-node n n2* |
*is-leveln-node* (*Suc n*) (*Node3 - n1 n2 n3*) $\longleftrightarrow$
*is-leveln-node n n1* $\wedge$ *is-leveln-node n n2* $\wedge$ *is-leveln-node n n3* |
*is-leveln-node - -* $\longleftrightarrow$ *False*

**primrec** *is-leveln-digit* :: *nat* $\Rightarrow$ $('e,'a)$ *Digit* $\Rightarrow$ *bool* **where**
*is-leveln-digit n* (*One n1*) $\longleftrightarrow$ *is-leveln-node n n1* |
*is-leveln-digit n* (*Two n1 n2*) $\longleftrightarrow$ *is-leveln-node n n1* $\wedge$
*is-leveln-node n n2* |
*is-leveln-digit n* (*Three n1 n2 n3*) $\longleftrightarrow$ *is-leveln-node n n1* $\wedge$

```
  is-leveln-node n n2 ∧ is-leveln-node n n3 |
 is-leveln-digit n (Four n1 n2 n3 n4) ⟷ is-leveln-node n n1 ∧
  is-leveln-node n n2 ∧ is-leveln-node n n3 ∧ is-leveln-node n n4

primrec is-leveln-ftree :: nat ⇒ ('e,'a) FingerTreeStruc ⇒ bool where
 is-leveln-ftree n Empty ⟷ True |
 is-leveln-ftree n (Single nd) ⟷ is-leveln-node n nd |
 is-leveln-ftree n (Deep - l t r) ⟷ is-leveln-digit n l ∧
  is-leveln-digit n r ∧ is-leveln-ftree (Suc n) t

primrec is-measured-node :: ('e,'a::monoid-add) Node ⇒ bool where
 is-measured-node (Tip - -) ⟷ True |
 is-measured-node (Node2 a n1 n2) ⟷ ((is-measured-node n1) ∧
  (is-measured-node n2)) ∧ (a = (gmn n1) + (gmn n2)) |
 is-measured-node (Node3 a n1 n2 n3) ⟷ ((is-measured-node n1) ∧
  (is-measured-node n2) ∧ (is-measured-node n3)) ∧
  (a = (gmn n1) + (gmn n2) + (gmn n3))

primrec is-measured-digit :: ('e,'a::monoid-add) Digit ⇒ bool where
 is-measured-digit (One a) = is-measured-node a |
 is-measured-digit (Two a b) =
  ((is-measured-node a) ∧ (is-measured-node b))|
 is-measured-digit (Three a b c) =
  ((is-measured-node a) ∧ (is-measured-node b) ∧ (is-measured-node c))|
 is-measured-digit (Four a b c d) = ((is-measured-node a) ∧
  (is-measured-node b) ∧ (is-measured-node c) ∧ (is-measured-node d))

primrec is-measured-ftree :: ('e,'a::monoid-add) FingerTreeStruc ⇒ bool where
 is-measured-ftree Empty ⟷ True |
 is-measured-ftree (Single n1) ⟷ (is-measured-node n1) |
 is-measured-ftree (Deep a l m r) ⟷ ((is-measured-digit l) ∧
  (is-measured-ftree m) ∧ (is-measured-digit r)) ∧
  (a = ((gmd l) + (gmft m) + (gmd r)))
```

Structural invariant for finger trees

```
definition ft-invar t == is-leveln-ftree 0 t ∧ is-measured-ftree t
```

### 1.1.2 Abstraction to Lists

```
primrec nodeToList :: ('e,'a) Node ⇒ ('e × 'a) list where
 nodeToList (Tip e a) = [(e,a)]|
 nodeToList (Node2 - a b) = (nodeToList a) @ (nodeToList b)|
 nodeToList (Node3 - a b c)
  = (nodeToList a) @ (nodeToList b) @ (nodeToList c)

primrec digitToList :: ('e,'a) Digit ⇒ ('e × 'a) list where
 digitToList (One a) = nodeToList a|
 digitToList (Two a b) = (nodeToList a) @ (nodeToList b)|
 digitToList (Three a b c)
```

$= (nodeToList\ a)\ @\ (nodeToList\ b)\ @\ (nodeToList\ c)|$
$digitToList\ (Four\ a\ b\ c\ d)$
$= (nodeToList\ a)\ @\ (nodeToList\ b)\ @\ (nodeToList\ c)\ @\ (nodeToList\ d)$

List representation of a finger tree

**primrec** $toList :: ('e, 'a)\ FingerTreeStruc \Rightarrow ('e \times 'a)\ list$ **where**
$toList\ Empty = []|$
$toList\ (Single\ a) = nodeToList\ a|$
$toList\ (Deep\ -\ pr\ m\ sf) = (digitToList\ pr)\ @\ (toList\ m)\ @\ (digitToList\ sf)$

**lemma** *nodeToList-empty*: $nodeToList\ nd \neq Nil$
**by** (*induct nd*) *auto*

**lemma** *digitToList-empty*: $digitToList\ d \neq Nil$
**by** (*cases d*, *auto simp add*: *nodeToList-empty*)

Auxiliary lemmas

**lemma** *gmn-correct*:
**assumes** *is-measured-node nd*
**shows** $gmn\ nd = sum\text{-}list\ (map\ snd\ (nodeToList\ nd))$
**by** (*insert assms*, *induct nd*) (*auto simp add*: *add.assoc*)

**lemma** *gmd-correct*:
**assumes** *is-measured-digit d*
**shows** $gmd\ d = sum\text{-}list\ (map\ snd\ (digitToList\ d))$
**by** (*insert assms*, *cases d*, *auto simp add*: *gmn-correct add.assoc*)

**lemma** *gmft-correct*: *is-measured-ftree t*
$\Longrightarrow (gmft\ t) = sum\text{-}list\ (map\ snd\ (toList\ t))$
**by** (*induct t*, *auto simp add*: *ft-invar-def gmd-correct gmn-correct add.assoc*)
**lemma** *gmft-correct2*: $ft\text{-}invar\ t \Longrightarrow (gmft\ t) = sum\text{-}list\ (map\ snd\ (toList\ t))$
**by** (*simp only*: *ft-invar-def gmft-correct*)

## 1.2 Operations

### 1.2.1 Empty tree

**lemma** *Empty-correct*[*simp*]:
$toList\ Empty = []$
*ft-invar Empty*
**by** (*simp-all add*: *ft-invar-def*)

Exactly the empty finger tree represents the empty list

**lemma** *toList-empty*: $toList\ t = [] \longleftrightarrow t = Empty$
**by** (*induct t*, *auto simp add*: *nodeToList-empty digitToList-empty*)

### 1.2.2 Annotation

Sum of annotations of all elements of a finger tree

**definition** *annot* :: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ '*a*
**where** *annot t* = *gmft t*

**lemma** *annot-correct*:
*ft-invar t* ⟹ *annot t* = *sum-list* (*map snd* (*toList t*))
**using** *gmft-correct*
**unfolding** *annot-def*
**by** (*simp add*: *gmft-correct2*)

### 1.2.3 Appending

Auxiliary functions to fill in the annotations

**definition** *deep*:: ('*e*,'*a*::*monoid-add*) *Digit* ⇒ ('*e*,'*a*) *FingerTreeStruc*
⇒ ('*e*,'*a*) *Digit* ⇒ ('*e*, '*a*) *FingerTreeStruc* **where**
*deep pr m sf* = *Deep* ((*gmd pr*) + (*gmft m*) + (*gmd sf*)) *pr m sf*
**definition** *node2* **where**
*node2 nd1 nd2* = *Node2* ((*gmn nd1*)+(*gmn nd2*)) *nd1 nd2*
**definition** *node3* **where**
*node3 nd1 nd2 nd3* = *Node3* ((*gmn nd1*)+(*gmn nd2*)+(*gmn nd3*)) *nd1 nd2 nd3*

Append a node at the left end

**fun** *nlcons* :: ('*e*,'*a*::*monoid-add*) *Node* ⇒ ('*e*,'*a*) *FingerTreeStruc*
⇒ ('*e*,'*a*) *FingerTreeStruc*
**where**
— Recursively we append a node, if the digit is full we push down a node3
*nlcons a Empty* = *Single a* |
*nlcons a* (*Single b*) = *deep* (*One a*) *Empty* (*One b*) |
*nlcons a* (*Deep* - (*One b*) *m sf*) = *deep* (*Two a b*) *m sf* |
*nlcons a* (*Deep* - (*Two b c*) *m sf*) = *deep* (*Three a b c*) *m sf* |
*nlcons a* (*Deep* - (*Three b c d*) *m sf*) = *deep* (*Four a b c d*) *m sf* |
*nlcons a* (*Deep* - (*Four b c d e*) *m sf*)
= *deep* (*Two a b*) (*nlcons* (*node3 c d e*) *m*) *sf*

Append a node at the right end

**fun** *nrcons* :: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc*
⇒ ('*e*,'*a*) *Node* ⇒ ('*e*,'*a*) *FingerTreeStruc* **where**
— Recursively we append a node, if the digit is full we push down a node3
*nrcons Empty a* = *Single a* |
*nrcons* (*Single b*) *a* = *deep* (*One b*) *Empty* (*One a*) |
*nrcons* (*Deep* - *pr m* (*One b*)) *a* = *deep pr m* (*Two b a*)|
*nrcons* (*Deep* - *pr m* (*Two b c*)) *a* = *deep pr m* (*Three b c a*) |
*nrcons* (*Deep* - *pr m* (*Three b c d*)) *a* = *deep pr m* (*Four b c d a*) |
*nrcons* (*Deep* - *pr m* (*Four b c d e*)) *a*
= *deep pr* (*nrcons m* (*node3 b c d*)) (*Two e a*)

**lemma** *nlcons-invlevel*: ⟦*is-leveln-ftree n t*; *is-leveln-node n nd*⟧
⟹ *is-leveln-ftree n* (*nlcons nd t*)
**by** (*induct t arbitrary*: *n nd rule*: *nlcons.induct*)

(*auto simp add*: *deep-def node3-def*)

**lemma** *nlcons-invmeas*: ⟦*is-measured-ftree t*; *is-measured-node nd*⟧
⟹ *is-measured-ftree* (*nlcons nd t*)
**by** (*induct t arbitrary*: *nd rule*: *nlcons.induct*)
(*auto simp add*: *deep-def node3-def*)

**lemmas** *nlcons-inv* = *nlcons-invlevel nlcons-invmeas*

**lemma** *nlcons-list*: *toList* (*nlcons a t*) = (*nodeToList a*) @ (*toList t*)
**apply** (*induct t arbitrary*: *a rule*: *nlcons.induct*)
**apply** (*auto simp add*: *deep-def toList-def node3-def*)
**done**

**lemma** *nrcons-invlevel*: ⟦*is-leveln-ftree n t*; *is-leveln-node n nd*⟧
⟹ *is-leveln-ftree n* (*nrcons t nd*)
**apply** (*induct t nd arbitrary*: *nd n rule*:*nrcons.induct*)
**apply**(*auto simp add*: *deep-def node3-def*)
**done**

**lemma** *nrcons-invmeas*: ⟦*is-measured-ftree t*; *is-measured-node nd*⟧
⟹ *is-measured-ftree* (*nrcons t nd*)
**apply** (*induct t nd arbitrary*: *nd rule*:*nrcons.induct*)
**apply**(*auto simp add*: *deep-def node3-def*)
**done**

**lemmas** *nrcons-inv* = *nrcons-invlevel nrcons-invmeas*

**lemma** *nrcons-list*: *toList* (*nrcons t a*) = (*toList t*) @ (*nodeToList a*)
**apply** (*induct t a arbitrary*: *a rule*: *nrcons.induct*)
**apply** (*auto simp add*: *deep-def toList-def node3-def*)
**done**

Append an element at the left end

**definition** *lcons* :: ('*e* × '*a*::*monoid-add*)
⇒ ('*e*,'*a*) *FingerTreeStruc* ⇒ ('*e*,'*a*) *FingerTreeStruc* (**infixr** ◁ *65*) **where**
*a* ◁ *t* = *nlcons* (*Tip* (*fst a*) (*snd a*)) *t*

**lemma** *lcons-correct*:
**assumes** *ft-invar t*
**shows** *ft-invar* (*a* ◁ *t*) **and** *toList* (*a* ◁ *t*) = *a* # (*toList t*)
**using** *assms*
**unfolding** *ft-invar-def*
**by** (*simp-all add*: *lcons-def nlcons-list nlcons-invlevel nlcons-invmeas*)

**lemma** *lcons-inv*:*ft-invar t* ⟹ *ft-invar* (*a* ◁ *t*)
**by** (*rule lcons-correct*)

**lemma** *lcons-list*[*simp*]: *toList* (*a* ◁ *t*) = *a* # (*toList t*)

**by** (*simp add*: *lcons-def nlcons-list*)

Append an element at the right end

**definition** *rcons*
:: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ ('*e* × '*a*) ⇒ ('*e*,'*a*) *FingerTreeStruc*
(**infixl** ▷ *65*) **where**
*t* ▷ *a* = *nrcons t* (*Tip* (*fst a*) (*snd a*))

**lemma** *rcons-correct*:
**assumes** *ft-invar t*
**shows** *ft-invar* (*t* ▷ *a*) **and** *toList* (*t* ▷ *a*) = (*toList t*) @ [*a*]
**using** *assms*
**by** (*auto simp add*: *nrcons-inv ft-invar-def rcons-def nrcons-list*)

**lemma** *rcons-inv*:*ft-invar t* ⟹ *ft-invar* (*t* ▷ *a*)
**by** (*rule rcons-correct*)

**lemma** *rcons-list*[*simp*]: *toList* (*t* ▷ *a*) = (*toList t*) @ [*a*]
**by**(*auto simp add*: *nrcons-list rcons-def*)

### 1.2.4 Convert list to tree

**primrec** *toTree* :: ('*e* × '*a*::*monoid-add*) *list* ⇒ ('*e*,'*a*) *FingerTreeStruc* **where**
*toTree* [] = *Empty*|
*toTree* (*a*#*xs*) = *a* ◁ (*toTree xs*)

**lemma** *toTree-correct*[*simp*]:
*ft-invar* (*toTree l*)
*toList* (*toTree l*) = *l*
**apply** (*induct l*)
**apply** (*simp add*: *ft-invar-def*)
**apply** *simp*
**apply** (*simp add*: *toTree-def lcons-list lcons-inv*)
**apply** (*simp add*: *toTree-def lcons-list lcons-inv*)
**done**

Note that this lemma is a completeness statement of our implementation, as it can be read as: ,,All lists of elements have a valid representation as a finger tree."

### 1.2.5 Detaching leftmost/rightmost element

**primrec** *digitToTree* :: ('*e*,'*a*::*monoid-add*) *Digit* ⇒ ('*e*,'*a*) *FingerTreeStruc*
**where**
*digitToTree* (*One a*) = *Single a*|
*digitToTree* (*Two a b*) = *deep* (*One a*) *Empty* (*One b*)|
*digitToTree* (*Three a b c*) = *deep* (*Two a b*) *Empty* (*One c*)|
*digitToTree* (*Four a b c d*) = *deep* (*Two a b*) *Empty* (*Two c d*)

**primrec** *nodeToDigit* :: ('*e*,'*a*) *Node* ⇒ ('*e*,'*a*) *Digit* **where**
*nodeToDigit* (*Tip e a*) = *One* (*Tip e a*)|
*nodeToDigit* (*Node2* - *a b*) = *Two a b*|
*nodeToDigit* (*Node3* - *a b c*) = *Three a b c*

**fun** *nlistToDigit* :: ('*e*,'*a*) *Node list* ⇒ ('*e*,'*a*) *Digit* **where**
*nlistToDigit* [*a*] = *One a* |
*nlistToDigit* [*a*,*b*] = *Two a b* |
*nlistToDigit* [*a*,*b*,*c*] = *Three a b c* |
*nlistToDigit* [*a*,*b*,*c*,*d*] = *Four a b c d*

**primrec** *digitToNlist* :: ('*e*,'*a*) *Digit* ⇒ ('*e*,'*a*) *Node list* **where**
*digitToNlist* (*One a*) = [*a*] |
*digitToNlist* (*Two a b*) = [*a*,*b*] |
*digitToNlist* (*Three a b c*) = [*a*,*b*,*c*] |
*digitToNlist* (*Four a b c d*) = [*a*,*b*,*c*,*d*]

Auxiliary function to unwrap a Node element

**primrec** *n-unwrap*:: ('*e*,'*a*) *Node* ⇒ ('*e* × '*a*) **where**
*n-unwrap* (*Tip e a*) = (*e*,*a*)|
*n-unwrap* (*Node2* - *a b*) = *undefined*|
*n-unwrap* (*Node3* - *a b c*) = *undefined*

**type-synonym** ('*e*,'*a*) *ViewnRes* = (('*e*,'*a*) *Node* × ('*e*,'*a*) *FingerTreeStruc*) *option*
**lemma** *viewnres-cases*:
**fixes** *r* :: ('*e*,'*a*) *ViewnRes*
**obtains** (*Nil*) *r*=*None* |
(*Cons*) *a t* **where** *r*=*Some* (*a*,*t*)
**by** (*cases r*) *auto*

**lemma** *viewnres-split*:
*P* (*case-option f1* (*case-prod f2*) *x*) =
((*x* = *None* ⟶ *P f1*) ∧ (∀ *a b*. *x* = *Some* (*a*,*b*) ⟶ *P* (*f2 a b*)))
**by** (*auto split*: *option.split prod.split*)

Detach the leftmost node. Return *None* on empty finger tree.

**fun** *viewLn* :: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ ('*e*,'*a*) *ViewnRes* **where**
*viewLn Empty* = *None*|
*viewLn* (*Single a*) = *Some* (*a*, *Empty*)|
*viewLn* (*Deep* - (*Two a b*) *m sf*) = *Some* (*a*, (*deep* (*One b*) *m sf*))|
*viewLn* (*Deep* - (*Three a b c*) *m sf*) = *Some* (*a*, (*deep* (*Two b c*) *m sf*))|
*viewLn* (*Deep* - (*Four a b c d*) *m sf*) = *Some* (*a*, (*deep* (*Three b c d*) *m sf*))|
*viewLn* (*Deep* - (*One a*) *m sf*) =
(*case viewLn m of*
*None* ⇒ *Some* (*a*, (*digitToTree sf*)) |
*Some* (*b*, *m2*) ⇒ *Some* (*a*, (*deep* (*nodeToDigit b*) *m2 sf*)))

Detach the rightmost node. Return *None* on empty finger tree.

```
fun viewRn :: ('e,'a::monoid-add) FingerTreeStruc ⇒ ('e,'a) ViewnRes where
  viewRn Empty = None |
  viewRn (Single a) = Some (a, Empty) |
  viewRn (Deep - pr m (Two a b)) = Some (b, (deep pr m (One a))) |
  viewRn (Deep - pr m (Three a b c)) = Some (c, (deep pr m (Two a b))) |
  viewRn (Deep - pr m (Four a b c d)) = Some (d, (deep pr m (Three a b c))) |
  viewRn (Deep - pr m (One a)) =
    (case viewRn m of
      None ⇒ Some (a, (digitToTree pr))|
      Some (b, m2) ⇒ Some (a, (deep pr m2 (nodeToDigit b))))



lemma
  digitToTree-inv: is-leveln-digit n d ⟹ is-leveln-ftree n (digitToTree d)
  is-measured-digit d ⟹ is-measured-ftree (digitToTree d)
  apply (cases d,auto simp add: deep-def)
  apply (cases d,auto simp add: deep-def)
  done

lemma digitToTree-list: toList (digitToTree d) = digitToList d
  by (cases d) (auto simp add: deep-def)

lemma nodeToDigit-inv:
  is-leveln-node (Suc n) nd ⟹ is-leveln-digit n (nodeToDigit nd)
  is-measured-node nd ⟹ is-measured-digit (nodeToDigit nd)
  by (cases nd, auto) (cases nd, auto)

lemma nodeToDigit-list: digitToList (nodeToDigit nd) = nodeToList nd
  by (cases nd,auto)



lemma viewLn-empty: t ≠ Empty ⟷ (viewLn t) ≠ None
proof (cases t)
  case Empty thus ?thesis by simp
next
  case (Single Node) thus ?thesis by simp
next
  case (Deep a l x r) thus ?thesis
  apply(auto)
  apply(case-tac l)
  apply(auto)
  apply(cases viewLn x)
  apply(auto)
  done
qed

lemma viewLn-inv: ⟦
```

```
  is-measured-ftree t; is-leveln-ftree n t; viewLn t = Some (nd, s)
  ⟧ ⟹ is-measured-ftree s ∧ is-measured-node nd ∧
       is-leveln-ftree n s ∧ is-leveln-node n nd
  apply(induct t arbitrary: n nd s rule: viewLn.induct)
  apply(simp add: viewLn-empty)
  apply(simp)
  apply(auto simp add: deep-def)[1]
  apply(auto simp add: deep-def)[1]
  apply(auto simp add: deep-def)[1]
proof −
  fix ux a m sf n nd s
  assume av: ⋀n nd s.
         ⟦is-measured-ftree m; is-leveln-ftree n m; viewLn m = Some (nd, s)⟧
         ⟹ is-measured-ftree s ∧
            is-measured-node nd ∧ is-leveln-ftree n s ∧ is-leveln-node n nd
        is-measured-ftree (Deep ux (One a) m sf)
       is-leveln-ftree n (Deep ux (One a) m sf)
       viewLn (Deep ux (One a) m sf) = Some (nd, s)
  thus is-measured-ftree s ∧
        is-measured-node nd ∧ is-leveln-ftree n s ∧ is-leveln-node n nd
  proof (cases viewLn m rule: viewnres-cases)
    case Nil
    with av(4) have v1: nd = a s = digitToTree sf
    by auto
    from v1 av(2,3) show is-measured-ftree s ∧
      is-measured-node nd ∧ is-leveln-ftree n s ∧ is-leveln-node n nd
      apply(auto)
      apply(auto simp add: digitToTree-inv)
    done
  next
    case (Cons b m2)
    with av(4) have v2: nd = a s = (deep (nodeToDigit b) m2 sf)
    apply (auto simp add: deep-def)
    done
    note myiv = av(1)[of Suc n b m2]
    from v2 av(2,3) have is-measured-ftree m ∧ is-leveln-ftree (Suc n) m
    apply(simp)
    done
    hence bv: is-measured-ftree m2 ∧
   is-measured-node b ∧ is-leveln-ftree (Suc n) m2 ∧ is-leveln-node (Suc n) b
    using myiv Cons
    apply(simp)
    done
    with av(2,3) v2 show is-measured-ftree s ∧
         is-measured-node nd ∧ is-leveln-ftree n s ∧ is-leveln-node n nd
    apply(auto simp add: deep-def nodeToDigit-inv)
    done
  qed
qed
```

**lemma** *viewLn-list*: *viewLn t* = *Some* (*nd*, *s*)
⟹ *toList t* = (*nodeToList nd*) @ (*toList s*)
**apply**(*induct t arbitrary*: *nd s rule*: *viewLn.induct*)
**apply**(*simp*)
**apply**(*simp*)
**apply**(*simp*)
**apply**(*simp add*: *deep-def*)
**apply**(*auto simp add*: *toList-def*)[*1*]
**apply**(*simp*)
**apply**(*simp add*: *deep-def*)
**apply**(*auto simp add*: *toList-def*)[*1*]
**apply**(*simp*)
**apply**(*simp add*: *deep-def*)
**apply**(*auto simp add*: *toList-def*)[*1*]
**apply**(*simp*)
**subgoal premises** *prems* **for** *a m sf nd s*
**using** *prems*
**proof** (*cases viewLn m rule*: *viewnres-cases*)
**case** *Nil*
**hence** *av*: *m* = *Empty* **by** (*metis viewLn-empty*)
**from** *av prems*
**show** *nodeToList a* @ *toList m* @ *digitToList sf* = *nodeToList nd* @ *toList s*
**by** (*auto simp add*: *digitToTree-list*)
**next**
**case** (*Cons b m2*)
**with** *prems* **have** *bv*: *nd* = *a s* = (*deep* (*nodeToDigit b*) *m2 sf*)
**by** (*auto simp add*: *deep-def*)
**with** *Cons prems*
**show** *nodeToList a* @ *toList m* @ *digitToList sf* = *nodeToList nd* @ *toList s*
**apply**(*simp*)
**apply**(*simp add*: *deep-def*)
**apply**(*simp add*: *deep-def nodeToDigit-list*)
**done**
**qed**
**done**

**lemma** *viewRn-empty*: *t* ≠ *Empty* ⟷ (*viewRn t*) ≠ *None*
**proof** (*cases t*)
**case** *Empty* **thus** *?thesis* **by** *simp*
**next**
**case** (*Single Node*) **thus** *?thesis* **by** *simp*
**next**
**case** (*Deep a l x r*) **thus** *?thesis*
**apply**(*auto*)
**apply**(*case-tac r*)
**apply**(*auto*)
**apply**(*cases viewRn x*)
**apply**(*auto*)

```
  done
qed

lemma viewRn-inv: ⟦
  is-measured-ftree t; is-leveln-ftree n t; viewRn t = Some (nd, s)
  ⟧ ⟹ is-measured-ftree s ∧ is-measured-node nd ∧
      is-leveln-ftree n s ∧ is-leveln-node n nd
  apply(induct t arbitrary: n nd s rule: viewRn.induct)
  apply(simp add: viewRn-empty)
  apply(simp)
  apply(auto simp add: deep-def)[1]
  apply(auto simp add: deep-def)[1]
  apply(auto simp add: deep-def)[1]
proof −
  fix ux a m pr n nd s
  assume av: ⋀n nd s.
         ⟦is-measured-ftree m; is-leveln-ftree n m; viewRn m = Some (nd, s)⟧
         ⟹ is-measured-ftree s ∧
            is-measured-node nd ∧ is-leveln-ftree n s ∧ is-leveln-node n nd
        is-measured-ftree (Deep ux pr m (One a))
       is-leveln-ftree n (Deep ux pr m (One a))
       viewRn (Deep ux pr m (One a)) = Some (nd, s)
  thus is-measured-ftree s ∧
        is-measured-node nd ∧ is-leveln-ftree n s ∧ is-leveln-node n nd
  proof (cases viewRn m rule: viewnres-cases)
    case Nil
    with av(4) have v1: nd = a s = digitToTree pr
    by auto
    from v1 av(2,3) show is-measured-ftree s ∧
      is-measured-node nd ∧ is-leveln-ftree n s ∧ is-leveln-node n nd
      apply(auto)
      apply(auto simp add: digitToTree-inv)
    done
  next
    case (Cons b m2)
    with av(4) have v2: nd = a s = (deep pr m2 (nodeToDigit b))
    apply (auto simp add: deep-def)
    done
    note myiv = av(1)[of Suc n b m2]
    from v2 av(2,3) have is-measured-ftree m ∧ is-leveln-ftree (Suc n) m
    apply(simp)
    done
    hence bv: is-measured-ftree m2 ∧
   is-measured-node b ∧ is-leveln-ftree (Suc n) m2 ∧ is-leveln-node (Suc n) b
    using myiv Cons
    apply(simp)
    done
    with av(2,3) v2 show is-measured-ftree s ∧
         is-measured-node nd ∧ is-leveln-ftree n s ∧ is-leveln-node n nd
```

```
    apply(auto simp add: deep-def nodeToDigit-inv)
    done
  qed
qed

lemma viewRn-list: viewRn t = Some (nd, s)
  ⟹ toList t = (toList s) @ (nodeToList nd)
  apply(induct t arbitrary: nd s rule: viewRn.induct)
  apply(simp)
  apply(simp)
  apply(simp)
  apply(simp add: deep-def)
  apply(auto simp add: toList-def)[1]
  apply(simp)
  apply(simp add: deep-def)
  apply(auto simp add: toList-def)[1]
  apply(simp)
  apply(simp add: deep-def)
  apply(auto simp add: toList-def)[1]
  apply(simp)
  subgoal premises prems for pr m a nd s
  proof (cases viewRn m rule: viewnres-cases)
    case Nil
    from Nil have av: m = Empty by (metis viewRn-empty)
    from av prems
    show digitToList pr @ toList m @ nodeToList a = toList s @ nodeToList nd
      by (auto simp add: digitToTree-list)
  next
    case (Cons b m2)
    with prems have bv: nd = a s = (deep pr m2 (nodeToDigit b))
    apply(auto simp add: deep-def) done
    with Cons prems
    show digitToList pr @ toList m @ nodeToList a = toList s @ nodeToList nd
      apply(simp)
      apply(simp add: deep-def)
      apply(simp add: deep-def nodeToDigit-list)
     done
  qed
  done


type-synonym ('e,'a) viewres = (('e ×'a) × ('e,'a) FingerTreeStruc) option
```

Detach the leftmost element. Return *None* on empty finger tree.

```
definition viewL :: ('e,'a::monoid-add) FingerTreeStruc ⇒ ('e,'a) viewres
  where
viewL t = (case viewLn t of
  None ⇒ None |
  (Some (a, t2)) ⇒ Some ((n-unwrap a), t2))
```

**lemma** *viewL-correct*:
**assumes** *INV*: *ft-invar t*
**shows**
(*t=Empty* $\Longrightarrow$ *viewL t = None*)
(*t≠Empty* $\Longrightarrow$ ($\exists$ *a s. viewL t = Some* (*a*, *s*) $\wedge$ *ft-invar s*
$\wedge$ *toList t = a # toList s*))
**proof** −
**assume** *t=Empty* **thus** *viewL t = None* **by** (*simp add*: *viewL-def*)
**next**
**assume** *NE*: *t* $\neq$ *Empty*
**from** *INV* **have** *INV'*: *is-leveln-ftree 0 t is-measured-ftree t*
**by** (*simp-all add*: *ft-invar-def*)
**from** *NE* **have** *v1*: *viewLn t* $\neq$ *None* **by** (*auto simp add*: *viewLn-empty*)
**then obtain** *nd s* **where** *vn*: *viewLn t = Some* (*nd*, *s*)
**by** (*cases viewLn t*) (*auto*)
**from** *this* **obtain** *a* **where** *v1*: *viewL t = Some* (*a*, *s*)
**by** (*auto simp add*: *viewL-def*)
**from** *INV' vn* **have**
*v2*: *is-measured-ftree s* $\wedge$ *is-leveln-ftree 0 s*
$\wedge$ *is-leveln-node 0 nd* $\wedge$ *is-measured-node nd*
*toList t* = (*nodeToList nd*) @ (*toList s*)
**by** (*auto simp add*: *viewLn-inv*[*of t 0 nd s*] *viewLn-list*[*of t*])
**with** *v1 vn* **have** *v3*: *nodeToList nd* = [*a*]
**apply** (*auto simp add*: *viewL-def* )
**apply** (*induct nd*)
**apply** *auto*
**done**
**with** *v1 v2*
**show** $\exists$ *a s. viewL t = Some* (*a*, *s*) $\wedge$ *ft-invar s* $\wedge$ *toList t = a # toList s*
**by** (*auto simp add*: *ft-invar-def*)
**qed**

**lemma** *viewL-correct-empty*[*simp*]: *viewL Empty = None*
**by** (*simp add*: *viewL-def*)

**lemma** *viewL-correct-nonEmpty*:
**assumes** *ft-invar t t* $\neq$ *Empty*
**obtains** *a s* **where**
*viewL t = Some* (*a*, *s*) *ft-invar s toList t = a # toList s*
**using** *assms viewL-correct* **by** *blast*

Detach the rightmost element. Return *None* on empty finger tree.

**definition** *viewR* :: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ ('*e*,'*a*) *viewres*
**where**
*viewR t* = (*case viewRn t of*
*None* ⇒ *None* |
(*Some* (*a*, *t2*)) ⇒ *Some* ((*n-unwrap a*), *t2*))

```
lemma viewR-correct:
  assumes INV: ft-invar t
  shows
  (t = Empty ⟹ viewR t = None)
  (t ≠ Empty ⟹ (∃ a s. viewR t = Some (a, s) ∧ ft-invar s
                        ∧ toList t = toList s @ [a]))
proof −
  assume t=Empty thus viewR t = None by (simp add: viewR-def)
next
  assume NE: t ≠ Empty
  from INV have INV': is-leveln-ftree 0 t is-measured-ftree t
    unfolding ft-invar-def by simp-all
  from NE have v1: viewRn t ≠ None by (auto simp add: viewRn-empty)
  then obtain nd s where vn: viewRn t = Some (nd, s)
    by (cases viewRn t) (auto)
  from this obtain a where v1: viewR t = Some (a, s)
    by (auto simp add: viewR-def)
  from INV' vn have
    v2: is-measured-ftree s ∧ is-leveln-ftree 0 s
        ∧ is-leveln-node 0 nd ∧ is-measured-node nd
       toList t = (toList s) @ (nodeToList nd)
    by (auto simp add: viewRn-inv[of t 0 nd s] viewRn-list[of t])
  with v1 vn have v3: nodeToList nd = [a]
    apply (auto simp add: viewR-def )
    apply (induct nd)
    apply auto
    done
  with v1 v2
  show ∃ a s. viewR t = Some (a, s) ∧ ft-invar s ∧ toList t = toList s @ [a]
    unfolding ft-invar-def by auto
qed

lemma viewR-correct-empty[simp]: viewR Empty = None
  unfolding viewR-def by simp

lemma viewR-correct-nonEmpty:
  assumes ft-invar t t ≠ Empty
  obtains a s where
  viewR t = Some (a, s) ft-invar s ∧ toList t = toList s @ [a]
  using assms viewR-correct by blast
```

Finger trees viewed as a double-ended queue. The head and tail functions here are only defined for non-empty queues, while the view-functions were also defined for empty finger trees.

Check for emptiness

```
definition isEmpty :: ('e,'a) FingerTreeStruc ⇒ bool where
  [code del]: isEmpty t = (t = Empty)
lemma isEmpty-correct: isEmpty t ⟷ toList t = []
```

**unfolding** *isEmpty-def* **by** (*simp add*: *toList-empty*)
— Avoid comparison with (=), and thus unnecessary equality-class parameter on element types in generated code
**lemma** [*code*]: *isEmpty t* = (*case t of Empty* ⇒ *True* | - ⇒ *False*)
**apply** (*cases t*)
**apply** (*auto simp add*: *isEmpty-def*)
**done**

Leftmost element

**definition** *head* :: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ '*e* × '*a* **where**
*head t* = (*case viewL t of* (*Some* (*a*, -)) ⇒ *a*)
**lemma** *head-correct*:
**assumes** *ft-invar t t* ≠ *Empty*
**shows** *head t* = *hd* (*toList t*)
**proof** −
**from** *assms viewL-correct*
**obtain** *a s* **where**
*v1*:*viewL t* = *Some* (*a*, *s*) ∧ *ft-invar s* ∧ *toList t* = *a* # *toList s* **by** *blast*
**hence** *v2*: *head t* = *a* **by** (*auto simp add*: *head-def*)
**from** *v1* **have** *hd* (*toList t*) = *a* **by** *simp*
**with** *v2* **show** *?thesis* **by** *simp*
**qed**

All but the leftmost element

**definition** *tail*
:: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ ('*e*,'*a*) *FingerTreeStruc*
**where**
*tail t* = (*case viewL t of* (*Some* (-, *m*)) ⇒ *m*)
**lemma** *tail-correct*:
**assumes** *ft-invar t t* ≠ *Empty*
**shows** *toList* (*tail t*) = *tl* (*toList t*) **and** *ft-invar* (*tail t*)
**proof** −
**from** *assms viewL-correct*
**obtain** *a s* **where**
*v1*:*viewL t* = *Some* (*a*, *s*) ∧ *ft-invar s* ∧ *toList t* = *a* # *toList s* **by** *blast*
**hence** *v2*: *tail t* = *s* **by** (*auto simp add*: *tail-def*)
**from** *v1* **have** *tl* (*toList t*) = *toList s* **by** *simp*
**with** *v1 v2* **show**
*toList* (*tail t*) = *tl* (*toList t*)
*ft-invar* (*tail t*)
**by** *simp-all*
**qed**

Rightmost element

**definition** *headR* :: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ '*e* × '*a* **where**
*headR t* = (*case viewR t of* (*Some* (*a*, -)) ⇒ *a*)
**lemma** *headR-correct*:
**assumes** *ft-invar t t* ≠ *Empty*
**shows** *headR t* = *last* (*toList t*)

**proof** −
**from** *assms viewR-correct*
**obtain** *a s* **where**
*v1*:*viewR t* = *Some* (*a*, *s*) ∧ *ft-invar s* ∧ *toList t* = *toList s* @ [*a*] **by** *blast*
**hence** *v2*: *headR t* = *a* **by** (*auto simp add*: *headR-def*)
**with** *v1* **show** *?thesis* **by** *auto*
**qed**

All but the rightmost element

**definition** *tailR*
:: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ ('*e*,'*a*) *FingerTreeStruc*
**where**
*tailR t* = (*case viewR t of* (*Some* (-, *m*)) ⇒ *m*)
**lemma** *tailR-correct*:
**assumes** *ft-invar t t* ≠ *Empty*
**shows** *toList* (*tailR t*) = *butlast* (*toList t*) **and** *ft-invar* (*tailR t*)
**proof** −
**from** *assms viewR-correct*
**obtain** *a s* **where**
*v1*:*viewR t* = *Some* (*a*, *s*) ∧ *ft-invar s* ∧ *toList t* = *toList s* @ [*a*] **by** *blast*
**hence** *v2*: *tailR t* = *s* **by** (*auto simp add*: *tailR-def*)
**with** *v1* **show** *toList* (*tailR t*) = *butlast* (*toList t*) **and** *ft-invar* (*tailR t*)
**by** *auto*
**qed**

### 1.2.6 Concatenation

**primrec** *lconsNlist* :: ('*e*,'*a*::*monoid-add*) *Node list*
⇒ ('*e*,'*a*) *FingerTreeStruc* ⇒ ('*e*,'*a*) *FingerTreeStruc* **where**
*lconsNlist* [] *t* = *t* |
*lconsNlist* (*x*#*xs*) *t* = *nlcons x* (*lconsNlist xs t*)
**primrec** *rconsNlist* :: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc*
⇒ ('*e*,'*a*) *Node list* ⇒ ('*e*,'*a*) *FingerTreeStruc* **where**
*rconsNlist t* [] = *t* |
*rconsNlist t* (*x*#*xs*) = *rconsNlist* (*nrcons t x*) *xs*

**fun** *nodes* :: ('*e*,'*a*::*monoid-add*) *Node list* ⇒ ('*e*,'*a*) *Node list* **where**
*nodes* [*a*, *b*] = [*node2 a b*] |
*nodes* [*a*, *b*, *c*] = [*node3 a b c*] |
*nodes* [*a*,*b*,*c*,*d*] = [*node2 a b*, *node2 c d*] |
*nodes* (*a*#*b*#*c*#*xs*) = (*node3 a b c*) # (*nodes xs*)

Recursively we concatenate two FingerTreeStrucs while we keep the inner Nodes in a list

**fun** *app3* :: ('*e*,'*a*::*monoid-add*) *FingerTreeStruc* ⇒ ('*e*,'*a*) *Node list*
⇒ ('*e*,'*a*) *FingerTreeStruc* ⇒ ('*e*,'*a*) *FingerTreeStruc* **where**
*app3 Empty xs t* = *lconsNlist xs t* |
*app3 t xs Empty* = *rconsNlist t xs* |
*app3* (*Single x*) *xs t* = *nlcons x* (*lconsNlist xs t*) |

```
  app3 t xs (Single x) = nrcons (rconsNlist t xs) x |
  app3 (Deep - pr1 m1 sf1) ts (Deep - pr2 m2 sf2) =
    deep pr1 (app3 m1
      (nodes ((digitToNlist sf1) @ ts @ (digitToNlist pr2))) m2) sf2

lemma lconsNlist-inv:
  assumes is-leveln-ftree n t
  and is-measured-ftree t
  and ∀ x∈set xs. (is-leveln-node n x ∧ is-measured-node x)
  shows
  is-leveln-ftree n (lconsNlist xs t) ∧ is-measured-ftree (lconsNlist xs t)
  by (insert assms, induct xs, auto simp add: nlcons-invlevel nlcons-invmeas)

lemma rconsNlist-inv:
  assumes is-leveln-ftree n t
  and is-measured-ftree t
  and ∀ x∈set xs. (is-leveln-node n x ∧ is-measured-node x)
  shows
  is-leveln-ftree n (rconsNlist t xs) ∧ is-measured-ftree (rconsNlist t xs)
  by (insert assms, induct xs arbitrary: t,
      auto simp add: nrcons-invlevel nrcons-invmeas)

lemma nodes-inv:
  assumes ∀ x ∈ set ts. is-leveln-node n x ∧ is-measured-node x
  and length ts ≥ 2
  shows ∀ x ∈ set (nodes ts). is-leveln-node (Suc n) x ∧ is-measured-node x
proof (insert assms, induct ts rule: nodes.induct)
  case (1 a b)
  thus ?case by (simp add: node2-def)
next
  case (2 a b c)
  thus ?case by (simp add: node3-def)
next
  case (3 a b c d)
  thus ?case by (simp add: node2-def)
next
  case (4 a b c v vb vc)
  thus ?case by (simp add: node3-def)
next
  show ⟦∀ x∈set []. is-leveln-node n x ∧ is-measured-node x; 2 ≤ length []⟧
    ⟹ ∀ x∈set (nodes []). is-leveln-node (Suc n) x ∧ is-measured-node x
    by simp
next
  show
    ⋀v. ⟦∀ x∈set [v]. is-leveln-node n x ∧ is-measured-node x; 2 ≤ length [v]⟧
    ⟹ ∀ x∈set (nodes [v]). is-leveln-node (Suc n) x ∧ is-measured-node x
    by simp
qed
```

```
lemma nodes-inv2:
  assumes is-leveln-digit n sf1
  and is-measured-digit sf1
  and is-leveln-digit n pr2
  and is-measured-digit pr2
  and ∀ x ∈ set ts. is-leveln-node n x ∧ is-measured-node x
  shows
  ∀x∈set (nodes (digitToNlist sf1 @ ts @ digitToNlist pr2)).
              is-leveln-node (Suc n) x ∧ is-measured-node x
proof −
  have v1: ∀x∈set (digitToNlist sf1 @ ts @ digitToNlist pr2).
                is-leveln-node n x ∧ is-measured-node x
    using assms
    apply (simp add: digitToNlist-def)
    apply (cases sf1)
    apply (cases pr2)
    apply simp-all
    apply (cases pr2)
    apply (simp-all)
    apply (cases pr2)
    apply (simp-all)
    apply (cases pr2)
    apply (simp-all)
    done
  have v2: length (digitToNlist sf1 @ ts @ digitToNlist pr2) ≥ 2
    apply (cases sf1)
    apply (cases pr2)
    apply simp-all
    done
  thus ?thesis
    using v1 nodes-inv[of digitToNlist sf1 @ ts @ digitToNlist pr2]
    by blast
qed

lemma app3-inv:
  assumes is-leveln-ftree n t1
  and is-leveln-ftree n t2
  and is-measured-ftree t1
  and is-measured-ftree t2
  and ∀ x∈set xs. (is-leveln-node n x ∧ is-measured-node x)
  shows is-leveln-ftree n (app3 t1 xs t2) ∧ is-measured-ftree (app3 t1 xs t2)
proof (insert assms, induct t1 xs t2 arbitrary: n rule: app3.induct)
  case (1 xs t n)
  thus ?case using lconsNlist-inv by simp
next
  case 2-1
  thus ?case by (simp add: rconsNlist-inv)
next
  case 2-2
```

```
  thus ?case by (simp add: lconsNlist-inv rconsNlist-inv)
next
  case 3-1
  thus ?case by (simp add: lconsNlist-inv nlcons-invlevel nlcons-invmeas )
next
  case 3-2
  thus ?case
    by (simp only: app3.simps)
       (simp add: lconsNlist-inv nlcons-invlevel nlcons-invmeas)
next
  case 4
  thus ?case
    by (simp only: app3.simps)
       (simp add: rconsNlist-inv nrcons-invlevel nrcons-invmeas)
next
  case (5 uu pr1 m1 sf1 ts uv pr2 m2 sf2 n)
  thus ?case
  proof −
    have v1: is-leveln-ftree (Suc n) m1
      and v2: is-leveln-ftree (Suc n) m2
      using 5.prems by (simp-all add: is-leveln-ftree-def)
    have v3: is-measured-ftree m1
      and v4: is-measured-ftree m2
      using 5.prems by (simp-all add: is-measured-ftree-def)
    have v5: is-leveln-digit n sf1
      is-measured-digit sf1
      is-leveln-digit n pr2
      is-measured-digit pr2
      ∀ x∈set ts. is-leveln-node n x ∧ is-measured-node x
      using 5.prems
      by (simp-all add: is-leveln-ftree-def is-measured-ftree-def)
    note v6 = nodes-inv2[OF v5]
    note v7 = 5.hyps[OF v1 v2 v3 v4 v6]
    have v8: is-leveln-digit n sf2
      is-measured-digit sf2
      is-leveln-digit n pr1
      is-measured-digit pr1
      using 5.prems
      by (simp-all add: is-leveln-ftree-def is-measured-ftree-def)

    show ?thesis using v7 v8
      by (simp add: is-leveln-ftree-def is-measured-ftree-def deep-def)
  qed
qed

primrec nlistToList:: (('e, 'a) Node) list ⇒ ('e × 'a) list where
  nlistToList [] = []|
  nlistToList (x#xs) = (nodeToList x) @ (nlistToList xs)
```

**lemma** *nodes-list*: *length xs* $\geq$ *2* $\Longrightarrow$ *nlistToList* (*nodes xs*) = *nlistToList xs*
**by** (*induct xs rule*: *nodes.induct*) (*auto simp add*: *node2-def node3-def*)

**lemma** *nlistToList-app*:
*nlistToList* (*xs*@*ys*) = (*nlistToList xs*) @ (*nlistToList ys*)
**by** (*induct xs arbitrary*: *ys*, *simp-all*)

**lemma** *nlistListLCons*: *toList* (*lconsNlist xs t*) = (*nlistToList xs*) @ (*toList t*)
**by** (*induct xs*) (*auto simp add*: *nlcons-list*)

**lemma** *nlistListRCons*: *toList* (*rconsNlist t xs*) = (*toList t*) @ (*nlistToList xs*)
**by** (*induct xs arbitrary*: *t*) (*auto simp add*: *nrcons-list*)

**lemma** *app3-list-lem1*:
*nlistToList* (*nodes* (*digitToNlist sf1* @ *ts* @ *digitToNlist pr2*)) =
*digitToList sf1* @ *nlistToList ts* @ *digitToList pr2*
**proof** −
**have** *len1*: *length* (*digitToNlist sf1* @ *ts* @ *digitToNlist pr2*) $\geq$ *2*
**by** (*cases sf1*,*cases pr2*,*simp-all*)

**have** (*nlistToList* (*digitToNlist sf1* @ *ts* @ *digitToNlist pr2*))
= (*digitToList sf1* @ *nlistToList ts* @ *digitToList pr2*)
**apply** (*cases sf1*, *cases pr2*)
**apply** (*simp-all add*: *nlistToList-app*)
**apply** (*cases pr2*, *auto*)
**apply** (*cases pr2*, *auto*)
**apply** (*cases pr2*, *auto*)
**done**
**with** *nodes-list*[*OF len1*] **show** *?thesis* **by** *simp*
**qed**

**lemma** *app3-list*:
*toList* (*app3 t1 xs t2*) = (*toList t1*) @ (*nlistToList xs*) @ (*toList t2*)
**apply** (*induct t1 xs t2 rule*: *app3.induct*)
**apply** (*simp-all add*: *nlistListLCons nlistListRCons nlcons-list nrcons-list*)
**apply** (*simp add*: *app3-list-lem1 deep-def*)
**done**

**definition** *app*
:: (*'e*,*'a*::*monoid-add*) *FingerTreeStruc* ⇒ (*'e*,*'a*) *FingerTreeStruc*
⇒ (*'e*,*'a*) *FingerTreeStruc*
**where** *app t1 t2* = *app3 t1* [] *t2*

**lemma** *app-correct*:
**assumes** *ft-invar t1 ft-invar t2*
**shows** *toList* (*app t1 t2*) = (*toList t1*) @ (*toList t2*)
**and** *ft-invar* (*app t1 t2*)

**using** *assms*
**by** (*auto simp add*: *app3-inv app3-list ft-invar-def app-def*)

**lemma** *app-inv*: ⟦*ft-invar t1*;*ft-invar t2*⟧ ⟹ *ft-invar* (*app t1 t2*)
**by** (*auto simp add*: *app3-inv ft-invar-def app-def*)

**lemma** *app-list*[*simp*]: *toList* (*app t1 t2*) = (*toList t1*) @ (*toList t2*)
**by** (*simp add*: *app3-list app-def*)

### 1.2.7 Splitting

**type-synonym** ('*e*,'*a*) *SplitDigit* =
('*e*,'*a*) *Node list* × ('*e*,'*a*) *Node* × ('*e*,'*a*) *Node list*
**type-synonym** ('*e*,'*a*) *SplitTree* =
('*e*,'*a*) *FingerTreeStruc* × ('*e*,'*a*) *Node* × ('*e*,'*a*) *FingerTreeStruc*

Auxiliary functions to create a correct finger tree even if the left or right digit is empty

**fun** *deepL* :: ('*e*,'*a*::*monoid-add*) *Node list* ⇒ ('*e*,'*a*) *FingerTreeStruc*
⇒ ('*e*,'*a*) *Digit* ⇒ ('*e*,'*a*) *FingerTreeStruc* **where**
*deepL* [] *m sf* = (*case* (*viewLn m*) *of None* ⇒ *digitToTree sf* |
(*Some* (*a*, *m2*)) ⇒ *deep* (*nodeToDigit a*) *m2 sf*) |
*deepL pr m sf* = *deep* (*nlistToDigit pr*) *m sf*
**fun** *deepR* :: ('*e*,'*a*::*monoid-add*) *Digit* ⇒ ('*e*,'*a*) *FingerTreeStruc*
⇒ ('*e*,'*a*) *Node list* ⇒ ('*e*,'*a*) *FingerTreeStruc* **where**
*deepR pr m* [] = (*case* (*viewRn m*) *of None* ⇒ *digitToTree pr* |
(*Some* (*a*, *m2*)) ⇒ *deep pr m2* (*nodeToDigit a*)) |
*deepR pr m sf* = *deep pr m* (*nlistToDigit sf*)

Splitting a list of nodes

**fun** *splitNlist* :: ('*a*::*monoid-add* ⇒ *bool*) ⇒ '*a* ⇒ ('*e*,'*a*) *Node list*
⇒ ('*e*,'*a*) *SplitDigit* **where**
*splitNlist p i* [*a*] = ([],*a*,[]) |
*splitNlist p i* (*a*#*b*) =
(*let i2* = (*i* + *gmn a*) *in*
(*if* (*p i2*)
*then* ([],*a*,*b*)
*else*
(*let* (*l*,*x*,*r*) = (*splitNlist p i2 b*) *in* ((*a*#*l*),*x*,*r*))))

Splitting a digit by converting it into a list of nodes

**definition** *splitDigit* :: ('*a*::*monoid-add* ⇒ *bool*) ⇒ '*a* ⇒ ('*e*,'*a*) *Digit*
⇒ ('*e*,'*a*) *SplitDigit* **where**
*splitDigit p i d* = *splitNlist p i* (*digitToNlist d*)

Creating a finger tree from list of nodes

**definition** *nlistToTree* :: ('*e*,'*a*::*monoid-add*) *Node list*
⇒ ('*e*,'*a*) *FingerTreeStruc* **where**

```
  nlistToTree xs = lconsNlist xs Empty
```

Recursive splitting into a left and right tree and a center node

```
fun nsplitTree :: ('a::monoid-add ⇒ bool) ⇒ 'a ⇒ ('e,'a) FingerTreeStruc
    ⇒ ('e,'a) SplitTree where
  nsplitTree p i Empty = (Empty, Tip undefined undefined, Empty)
     — Making the function total |
  nsplitTree p i (Single ea) = (Empty,ea,Empty) |
  nsplitTree p i (Deep - pr m sf) =
     (let
       vpr = (i + gmd pr);
       vm = (vpr + gmft m)
      in
       if (p vpr) then
         (let (l,x,r) = (splitDigit p i pr) in
           (nlistToTree l,x,deepL r m sf))
       else (if (p vm) then
         (let (ml,xs,mr) = (nsplitTree p vpr m);
           (l,x,r) = (splitDigit p (vpr + gmft ml) (nodeToDigit xs)) in
             (deepR pr ml l,x,deepL r mr sf))
       else
         (let (l,x,r) = (splitDigit p vm sf) in
           (deepR pr m l,x,nlistToTree r))
     ))


lemma nlistToTree-inv:
  ∀ x ∈ set nl. is-measured-node x ⟹ is-measured-ftree (nlistToTree nl)
  ∀ x ∈ set nl. is-leveln-node n x ⟹ is-leveln-ftree n (nlistToTree nl)
  by (unfold nlistToTree-def, induct nl, auto simp add: nlcons-invmeas)
     (induct nl, auto simp add: nlcons-invlevel)

lemma nlistToTree-list: toList (nlistToTree nl) = nlistToList nl
  by (auto simp add: nlistToTree-def nlistListLCons)

lemma deepL-inv:
  assumes is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
  and is-leveln-digit n sf ∧ is-measured-digit sf
  and ∀ x ∈ set pr. (is-measured-node x ∧ is-leveln-node n x) ∧ length pr ≤ 4
  shows is-leveln-ftree n (deepL pr m sf) ∧ is-measured-ftree (deepL pr m sf)
  apply (insert assms)
  apply (induct pr m sf rule: deepL.induct)
  apply (simp split: viewnres-split)
  apply auto[1]
  apply (simp-all add: digitToTree-inv deep-def)
proof −
  fix m sf Node FingerTreeStruc
  assume is-leveln-ftree (Suc n) m is-measured-ftree m
         is-leveln-digit n sf is-measured-digit sf
```

```
        viewLn m = Some (Node, FingerTreeStruc)
  thus is-leveln-digit n (nodeToDigit Node)
       ∧ is-leveln-ftree (Suc n) FingerTreeStruc
    by (simp add: viewLn-inv[of m Suc n Node FingerTreeStruc] nodeToDigit-inv)
next
  fix m sf Node FingerTreeStruc
  assume assms1:
    is-leveln-ftree (Suc n) m is-measured-ftree m
    is-leveln-digit n sf is-measured-digit sf
    viewLn m = Some (Node, FingerTreeStruc)
  thus is-measured-digit (nodeToDigit Node) ∧ is-measured-ftree FingerTreeStruc
    apply (auto simp only: viewLn-inv[of m Suc n Node FingerTreeStruc])
  proof −
    from assms1 have is-measured-node Node ∧ is-leveln-node (Suc n) Node
      by (simp add: viewLn-inv[of m Suc n Node FingerTreeStruc])
    thus is-measured-digit (nodeToDigit Node)
      by (auto simp add: nodeToDigit-inv)
  qed
next
  fix v va
  assume
    is-measured-node v ∧ is-leveln-node n (v:: ('a,'b) Node) ∧
    length (va::('a, 'b) Node list) ≤ 3 ∧
    (∀ x∈set va. is-measured-node x ∧ is-leveln-node n x ∧ length va ≤ 3)
  thus is-leveln-digit n (nlistToDigit (v # va))
       ∧ is-measured-digit (nlistToDigit (v # va))
    by(cases v#va rule: nlistToDigit.cases,simp-all)
qed



lemma nlistToDigit-list:
  assumes 1 ≤ length xs ∧ length xs ≤ 4
  shows digitToList(nlistToDigit xs) = nlistToList xs
  by (insert assms, cases xs rule: nlistToDigit.cases,auto)

lemma deepL-list:
  assumes is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
  and is-leveln-digit n sf ∧ is-measured-digit sf
  and ∀ x ∈ set pr. (is-measured-node x ∧ is-leveln-node n x) ∧ length pr ≤ 4
  shows toList (deepL pr m sf) = nlistToList pr @ toList m @ digitToList sf
proof (insert assms, induct pr m sf rule: deepL.induct)
  case (1 m sf)
  thus ?case
  proof (auto split: viewnres-split simp add: deep-def)
    assume viewLn m = None
    hence m = Empty by (metis viewLn-empty)
    hence toList m = [] by simp
    thus toList (digitToTree sf) = toList m @ digitToList sf
```

```
    by (simp add:digitToTree-list)
  next
    fix nd t
    assume viewLn m = Some (nd, t)
      is-leveln-ftree (Suc n) m is-measured-ftree m
    hence nodeToList nd @ toList t = toList m by (metis viewLn-list)
    thus digitToList (nodeToDigit nd) @ toList t = toList m
      by (simp add: nodeToDigit-list)
  qed
next
  case (2 v va m sf)
  thus ?case
    apply (unfold deepL.simps)
    apply (simp add: deep-def)
    apply (simp add: nlistToDigit-list)
    done
qed

lemma deepR-inv:
  assumes is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
  and is-leveln-digit n pr ∧ is-measured-digit pr
  and ∀ x ∈ set sf. (is-measured-node x ∧ is-leveln-node n x) ∧ length sf ≤ 4
  shows is-leveln-ftree n (deepR pr m sf) ∧ is-measured-ftree (deepR pr m sf)
  apply (insert assms)
  apply (induct pr m sf rule: deepR.induct)
  apply (simp split: viewnres-split)
  apply auto[1]
  apply (simp-all add: digitToTree-inv deep-def)
proof −
  fix m pr Node FingerTreeStruc
  assume is-leveln-ftree (Suc n) m is-measured-ftree m
        is-leveln-digit n pr is-measured-digit pr
        viewRn m = Some (Node, FingerTreeStruc)
  thus
    is-leveln-digit n (nodeToDigit Node)
    ∧ is-leveln-ftree (Suc n) FingerTreeStruc
    by (simp add: viewRn-inv[of m Suc n Node FingerTreeStruc] nodeToDigit-inv)
next
  fix m pr Node FingerTreeStruc
  assume assms1:
    is-leveln-ftree (Suc n) m is-measured-ftree m
    is-leveln-digit n pr is-measured-digit pr
    viewRn m = Some (Node, FingerTreeStruc)
  thus is-measured-ftree FingerTreeStruc ∧ is-measured-digit (nodeToDigit Node)
    apply (auto simp only: viewRn-inv[of m Suc n Node FingerTreeStruc])
  proof −
    from assms1 have is-measured-node Node ∧ is-leveln-node (Suc n) Node
      by (simp add: viewRn-inv[of m Suc n Node FingerTreeStruc])
    thus is-measured-digit (nodeToDigit Node)
```

```
    by (auto simp add: nodeToDigit-inv)
  qed
next
  fix v va
  assume
    is-measured-node v ∧ is-leveln-node n (v:: ('a,'b) Node) ∧
    length (va::('a, 'b) Node list) ≤ 3 ∧
    (∀ x∈set va. is-measured-node x ∧ is-leveln-node n x ∧ length va ≤ 3)
  thus is-leveln-digit n (nlistToDigit (v # va)) ∧
       is-measured-digit (nlistToDigit (v # va))
    by(cases v#va rule: nlistToDigit.cases,simp-all)
qed


lemma deepR-list:
  assumes is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
  and is-leveln-digit n pr ∧ is-measured-digit pr
  and ∀ x ∈ set sf. (is-measured-node x ∧ is-leveln-node n x) ∧ length sf ≤ 4
  shows toList (deepR pr m sf) = digitToList pr @ toList m @ nlistToList sf
proof (insert assms, induct pr m sf rule: deepR.induct)
  case (1 pr m)
  thus ?case
  proof (auto split: viewnres-split simp add: deep-def)
    assume viewRn m = None
    hence m = Empty by (metis viewRn-empty)
    hence toList m = [] by simp
    thus toList (digitToTree pr) = digitToList pr @ toList m
      by (simp add:digitToTree-list)
  next
    fix nd t
    assume viewRn m = Some (nd, t) is-leveln-ftree (Suc n) m
           is-measured-ftree m
    hence toList t @ nodeToList nd = toList m by (metis viewRn-list)
    thus toList t @ digitToList (nodeToDigit nd) = toList m
      by (simp add: nodeToDigit-list)
  qed
next
  case (2 pr m v va)
  thus ?case
    apply (unfold deepR.simps)
    apply (simp add: deep-def)
    apply (simp add: nlistToDigit-list)
    done
qed

primrec gmnl:: ('e, 'a::monoid-add) Node list ⇒ 'a where
gmnl [] = 0|
gmnl (x#xs) = gmn x + gmnl xs
```

```
lemma gmnl-correct:
  assumes ∀ x ∈ set xs. is-measured-node x
  shows  gmnl xs = sum-list (map snd (nlistToList xs))
  by (insert assms, induct xs) (auto simp add: add.assoc gmn-correct)

lemma splitNlist-correct: ⟦
  ⋀(a::'a) (b::'a). p a ⟹ p (a + b);
  ¬ p i;
  p (i + gmnl (nl ::('e,'a::monoid-add) Node list));
  splitNlist p i nl = (l, n, r)
  ⟧ ⟹
  ¬ p (i + (gmnl l))
  ∧
  p (i + (gmnl l) + (gmn n))
  ∧
  nl = l @ n # r

proof (induct p i nl arbitrary: l n r rule: splitNlist.induct)
  case 1 thus ?case by simp
next
  case (2 p i a v va l n r) note IV = this
  show ?case
  proof (cases p (i + (gmn a)))
    case True with IV show ?thesis by simp
  next
    case False note IV2 = this IV  thus ?thesis
    proof −
      obtain l1 n1 r1 where
        v1[simp]: splitNlist p (i + gmn a) (v # va) = (l1, n1, r1)
        by (cases splitNlist p (i + gmn a) (v # va), blast)
      note miv = IV2(2)[of i + gmn a l1 n1 r1]
      have v2:p (i + gmn a + gmnl (v # va))
        using IV2(5) by (simp add: add.assoc)
      note miv2 =  miv[OF - IV2(1) IV2(3) IV2(1)  v2 v1]
      have v3: a # l1 = l n1 = n r1 = r  using IV2 v1 by auto
      with miv2 have
        v4: ¬ p (i + gmn a + gmnl l1) ∧
            p (i + gmn a + gmnl l1 + gmn n1) ∧
            v # va = l1 @ n1 # r1
        by auto
      with v2 v3 show ?thesis
        by (auto simp add: add.assoc)
    qed
  qed
next
  case 3 thus ?case by simp
qed

lemma digitToNlist-inv:
```

*is-measured-digit d* $\Longrightarrow$ ($\forall$ *x* $\in$ *set* (*digitToNlist d*). *is-measured-node x*)
*is-leveln-digit n d* $\Longrightarrow$ ($\forall$ *x* $\in$ *set* (*digitToNlist d*). *is-leveln-node n x*)
**by** (*cases d*, *auto*)(*cases d*, *auto*)

**lemma** *gmnl-gmd*:
*is-measured-digit d* $\Longrightarrow$ *gmnl* (*digitToNlist d*) = *gmd d*
**by** (*cases d*, *auto simp add*: *add.assoc*)

**lemma** *gmn-gmd*:
*is-measured-node nd* $\Longrightarrow$ *gmd* (*nodeToDigit nd*) = *gmn nd*
**by** (*auto simp add*: *nodeToDigit-inv nodeToDigit-list gmn-correct gmd-correct*)

**lemma** *splitDigit-inv*:
⟦
$\bigwedge$(*a*::'*a*) (*b*::'*a*). *p a* $\Longrightarrow$ *p* (*a* + *b*);
¬ *p i*;
*is-measured-digit d*;
*is-leveln-digit n d*;
*p* (*i* + *gmd* (*d* ::('*e*,'*a*::*monoid-add*) *Digit*));
*splitDigit p i d* = (*l*, *nd*, *r*)
⟧ $\Longrightarrow$
¬ *p* (*i* + (*gmnl l*))
∧
*p* (*i* + (*gmnl l*) + (*gmn nd*))
∧
($\forall$ *x* $\in$ *set l*. (*is-measured-node x* ∧ *is-leveln-node n x*))
∧
($\forall$ *x* $\in$ *set r*. (*is-measured-node x* ∧ *is-leveln-node n x*))
∧
(*is-measured-node nd* ∧ *is-leveln-node n nd* )

**proof** −
**fix** *p i d n l nd r*
**assume** *assms*: $\bigwedge$*a b*. *p a* $\Longrightarrow$ *p* (*a* + *b*) ¬ *p i is-measured-digit d*
*p* (*i* + *gmd d*) *splitDigit p i d* = (*l*, *nd*, *r*)
*is-leveln-digit n d*
**from** *assms*(*3*, *4*) **have** *v1*: *p* (*i* + *gmnl* (*digitToNlist d*))
**by** (*simp add*: *gmnl-gmd*)
**note** *snc* = *splitNlist-correct* [*of p i digitToNlist d l nd r*]
**from** *assms*(*5*) **have** *v2*: *splitNlist p i* (*digitToNlist d*) = (*l*, *nd*, *r*)
**by** (*simp add*: *splitDigit-def*)
**note** *snc1* = *snc*[*OF assms*(*1*) *assms*(*2*) *v1 v2*]
**hence** *v3*: ¬ *p* (*i* + *gmnl l*) ∧ *p* (*i* + *gmnl l* + *gmn nd*) ∧
*digitToNlist d* = *l* @ *nd* # *r* **by** *auto*
**from** *assms*(*3*,*6*) **have**
*v4*: $\forall$ *x* $\in$ *set* (*digitToNlist d*). *is-measured-node x*
$\forall$ *x* $\in$ *set* (*digitToNlist d*). *is-leveln-node n x*
**by**(*auto simp add*: *digitToNlist-inv*)
**with** *v3* **have** *v5*: $\forall$ *x* $\in$ *set l*. (*is-measured-node x* ∧ *is-leveln-node n x*)

```
    ∀ x ∈ set r. (is-measured-node x ∧ is-leveln-node n x)
    is-measured-node nd ∧ is-leveln-node n nd by auto
  with v3 v5 show
    ¬ p (i + gmnl l) ∧ p (i + gmnl l + gmn nd) ∧
    (∀x∈set l. is-measured-node x ∧ is-leveln-node n x) ∧
    (∀x∈set r. is-measured-node x ∧ is-leveln-node n x) ∧
    is-measured-node nd ∧ is-leveln-node n nd
    by auto
qed


lemma splitDigit-inv':
  ⟦
  splitDigit p i d = (l, nd, r);
  is-measured-digit d;
  is-leveln-digit n d
  ⟧ ⟹
  (∀ x ∈ set l. (is-measured-node x ∧ is-leveln-node n x))
  ∧
  (∀ x ∈ set r. (is-measured-node x ∧ is-leveln-node n x))
  ∧
  (is-measured-node nd ∧ is-leveln-node n nd )

  apply (unfold splitDigit-def)
  apply (cases d)
  apply (auto split: if-split-asm simp add: Let-def)
  done


lemma splitDigit-list: splitDigit p i d = (l,n,r) ⟹
  (digitToList d) = (nlistToList l) @ (nodeToList n) @ (nlistToList r)
  ∧ length l ≤ 4 ∧ length r ≤ 4
  apply (unfold splitDigit-def)
  apply (cases d)
  apply (auto split: if-split-asm simp add: Let-def)
  done

lemma gmnl-gmft: ∀ x ∈ set nl. is-measured-node x ⟹
  gmft (nlistToTree nl) = gmnl nl
by (auto simp add: gmnl-correct[of nl] nlistToTree-list[of nl]
                   nlistToTree-inv[of nl]  gmft-correct[of nlistToTree nl])

lemma gmftR-gmnl:
  assumes is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
  and is-leveln-digit n pr ∧ is-measured-digit pr
  and ∀ x ∈ set sf. (is-measured-node x ∧ is-leveln-node n x) ∧ length sf ≤ 4
  shows gmft (deepR pr m sf) = gmd pr + gmft m + gmnl sf
proof−
  from assms have
```

```
    v1: toList (deepR pr m sf) = digitToList pr @ toList m @ nlistToList sf
    by (auto simp add: deepR-list)
  from assms have
    v2: is-measured-ftree (deepR pr m sf)
    by (auto simp add: deepR-inv)
  with v1 have
    v3: gmft (deepR pr m sf) =
        sum-list (map snd (digitToList pr @ toList m @ nlistToList sf))
    by (auto simp add: gmft-correct)
  have
    v4:gmd pr + gmft m + gmnl sf =
        sum-list (map snd (digitToList pr @ toList m @ nlistToList sf))
    by (auto simp add: gmd-correct gmft-correct gmnl-correct assms add.assoc)
  with v3 show ?thesis by simp
qed

lemma nsplitTree-invpres: ⟦
  is-leveln-ftree n (s:: ('e,'a::monoid-add) FingerTreeStruc);
  is-measured-ftree s;
  ¬ p i;
  p (i + (gmft s));
  (nsplitTree p i s) = (l, nd, r)⟧
  ⟹
  is-leveln-ftree n l
  ∧
  is-measured-ftree l
  ∧
  is-leveln-ftree n r
  ∧
  is-measured-ftree r
  ∧
  is-leveln-node n nd
  ∧
  is-measured-node nd

proof (induct p i s arbitrary: n l nd r rule: nsplitTree.induct)
  case 1
  thus ?case by auto
next
  case 2 thus ?case by auto
next
  case (3 p i uu pr m sf n l nd r)
  thus ?case
  proof (cases p (i + gmd pr))
    case True with 3 show ?thesis
    proof −
      obtain l1 x r1 where
        l1xr1: splitDigit p i pr = (l1,x,r1)
        by (cases splitDigit p i pr, blast)
```

```
    with True 3 have
      v1: l = nlistToTree l1 nd = x r = deepL r1 m sf by auto
    from l1xr1 have
      v2: digitToList pr = nlistToList l1 @ nodeToList x @ nlistToList r1
      length l1 ≤ 4 length r1 ≤ 4
      by (auto simp add: splitDigit-list)
    from 3(2,3) have
      pr-m-sf-inv: is-leveln-digit n pr ∧ is-measured-digit pr
      is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
      is-leveln-digit n sf ∧ is-measured-digit sf by simp-all
    with 3(4,5) pr-m-sf-inv(1) True l1xr1
      splitDigit-inv'[of p i pr l1 x r1 n] have
      l1-x-r1-inv:
      ∀ x ∈ set l1. (is-measured-node x ∧ is-leveln-node n x)
      ∀ x ∈ set r1. (is-measured-node x ∧ is-leveln-node n x)
      is-measured-node x ∧ is-leveln-node n x
      by auto
    from l1-x-r1-inv v1 v2(3) pr-m-sf-inv have
      ziel3: is-leveln-ftree n l ∧ is-measured-ftree l ∧
      is-leveln-ftree n r ∧ is-measured-ftree r ∧
      is-leveln-node n nd ∧ is-measured-node nd
      by (auto simp add: nlistToTree-inv deepL-inv)
    thus ?thesis by simp
  qed
next
  case False note case1 = this with 3 show ?thesis
  proof (cases p (i + gmd pr + gmft m))
    case False with case1 3 show ?thesis
    proof −
      obtain l1 x r1 where
        l1xr1: splitDigit p (i + gmd pr + gmft m) sf = (l1,x,r1)
        by (cases splitDigit p (i + gmd pr + gmft m) sf, blast)
      with case1 False 3 have
        v1: l = deepR pr m l1 nd = x r = nlistToTree r1 by auto
      from l1xr1 have
        v2: digitToList sf = nlistToList l1 @ nodeToList x @ nlistToList r1
        length l1 ≤ 4 length r1 ≤ 4
        by (auto simp add: splitDigit-list)
      from 3(2,3) have
        pr-m-sf-inv: is-leveln-digit n pr ∧ is-measured-digit pr
        is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
        is-leveln-digit n sf ∧ is-measured-digit sf by simp-all
      from 3 have
        v7: p (i + gmd pr + gmft m + gmd sf) by (auto simp add: add.assoc)
      with pr-m-sf-inv 3(4) pr-m-sf-inv(3) case1 False l1xr1
          splitDigit-inv'[of p i + gmd pr + gmft m sf l1 x r1 n]
      have l1-x-r1-inv:
        ∀ x ∈ set l1. (is-measured-node x ∧ is-leveln-node n x)
        ∀ x ∈ set r1. (is-measured-node x ∧ is-leveln-node n x)
```

```
    is-measured-node x ∧ is-leveln-node n x
    by auto
  from l1-x-r1-inv v1 v2(2) pr-m-sf-inv have
    ziel3: is-leveln-ftree n l ∧ is-measured-ftree l ∧
    is-leveln-ftree n r ∧ is-measured-ftree r ∧
    is-leveln-node n nd ∧ is-measured-node nd
    by (auto simp add: nlistToTree-inv deepR-inv)
  from ziel3 show ?thesis by simp
 qed
next
 case True with case1 3 show ?thesis
 proof −
  obtain l1 x r1 where
    l1-x-r1 :nsplitTree p (i + gmd pr) m = (l1, x, r1)
    by (cases nsplitTree p (i + gmd pr) m, blast)
  from 3(2,3) have
    pr-m-sf-inv: is-leveln-digit n pr ∧ is-measured-digit pr
    is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
    is-leveln-digit n sf ∧ is-measured-digit sf by simp-all
  with True case1
    3.hyps[of i + gmd pr i + gmd pr + gmft m Suc n l1 x r1]
    3(6) l1-x-r1
  have l1-x-r1-inv:
    is-leveln-ftree (Suc n) l1 ∧ is-measured-ftree l1
    is-leveln-ftree (Suc n) r1 ∧ is-measured-ftree r1
    is-leveln-node (Suc n) x ∧ is-measured-node x
    by auto
  obtain l2 x2 r2 where l2-x2-r2:
    splitDigit p (i + gmd pr + gmft l1) (nodeToDigit x) = (l2,x2,r2)
    by (cases splitDigit p (i + gmd pr + gmft l1) (nodeToDigit x),blast)
  from l1-x-r1-inv have
    ndx-inv: is-leveln-digit n (nodeToDigit x) ∧
    is-measured-digit (nodeToDigit x)
    by (auto simp add: nodeToDigit-inv gmn-gmd)
  note spdi = splitDigit-inv'[of p i + gmd pr + gmft l1
                              nodeToDigit x l2 x2 r2 n]
  from ndx-inv l1-x-r1-inv(1) l2-x2-r2 3(4) have
    l2-x2-r2-inv:
    ∀ x∈set l2. is-measured-node x ∧ is-leveln-node n x
    ∀ x∈set r2. is-measured-node x ∧ is-leveln-node n x
    is-measured-node x2 ∧ is-leveln-node n x2
    by (auto simp add: spdi)
  note spdl =  splitDigit-list[of p i + gmd pr + gmft l1
                               nodeToDigit x l2 x2 r2]
  from l2-x2-r2 have
    l2-x2-r2-list:
    digitToList (nodeToDigit x) =
      nlistToList l2 @ nodeToList x2 @ nlistToList r2
    length l2 ≤ 4 ∧ length r2 ≤ 4
```

```
        by (auto simp add: spdl)
      from case1 True 3(6) l1-x-r1 l2-x2-r2 have
        l-nd-r:
        l = deepR pr l1 l2
        nd = x2
        r = deepL r2 r1 sf
        by auto
      note dr1 = deepR-inv[OF l1-x-r1-inv(1) pr-m-sf-inv(1)]
      from dr1 l2-x2-r2-inv l2-x2-r2-list(2) l-nd-r have
        l-inv: is-leveln-ftree n l ∧ is-measured-ftree l
        by simp
      note dl1 = deepL-inv[OF l1-x-r1-inv(2) pr-m-sf-inv(3)]
      from dl1 l2-x2-r2-inv l2-x2-r2-list(2) l-nd-r have
        r-inv: is-leveln-ftree n r ∧ is-measured-ftree r
        by simp
      from l2-x2-r2-inv l-nd-r have
        nd-inv: is-leveln-node n nd ∧ is-measured-node nd
        by simp
      from l-inv r-inv nd-inv
      show ?thesis by simp
    qed
  qed
 qed
qed

lemma nsplitTree-correct: ⟦
  is-leveln-ftree n (s:: ('e,'a::monoid-add) FingerTreeStruc);
  is-measured-ftree s;
  ⋀(a::'a) (b::'a). p a ⟹ p (a + b);
  ¬ p i;
  p (i + (gmft s));
  (nsplitTree p i s) = (l, nd, r)⟧
  ⟹ (toList s) = (toList l) @ (nodeToList nd) @ (toList r)
  ∧
  ¬ p (i + (gmft l))
  ∧
  p (i + (gmft l) + (gmn nd))
  ∧
  is-leveln-ftree n l
  ∧
  is-measured-ftree l
  ∧
  is-leveln-ftree n r
  ∧
  is-measured-ftree r
  ∧
  is-leveln-node n nd
  ∧
  is-measured-node nd
```

**proof** (*induct p i s arbitrary*: *n l nd r rule*: *nsplitTree.induct*)
**case** *1*
**thus** *?case* **by** *auto*
**next**
**case** *2* **thus** *?case* **by** *auto*
**next**
**case** (*3 p i uu pr m sf n l nd r*)
**thus** *?case*
**proof** (*cases p* (*i* + *gmd pr*))
**case** *True* **with** *3* **show** *?thesis*
**proof** −
**obtain** *l1 x r1* **where**
*l1xr1*: *splitDigit p i pr* = (*l1*,*x*,*r1*)
**by** (*cases splitDigit p i pr*, *blast*)
**with** *True 3*(*7*) **have**
*v1*: *l* = *nlistToTree l1 nd* = *x r* = *deepL r1 m sf* **by** *auto*
**from** *l1xr1* **have**
*v2*: *digitToList pr* = *nlistToList l1* @ *nodeToList x* @ *nlistToList r1*
*length l1* ≤ *4 length r1* ≤ *4*
**by** (*auto simp add*: *splitDigit-list*)
**from** *3*(*2*,*3*) **have**
*pr-m-sf-inv*: *is-leveln-digit n pr* ∧ *is-measured-digit pr*
*is-leveln-ftree* (*Suc n*) *m* ∧ *is-measured-ftree m*
*is-leveln-digit n sf* ∧ *is-measured-digit sf* **by** *simp-all*
**with** *3*(*4*,*5*) *pr-m-sf-inv*(*1*) *True l1xr1*
*splitDigit-inv*[*of p i pr n l1 x r1*] **have**
*l1-x-r1-inv*:
¬ *p* (*i* + (*gmnl l1*))
*p* (*i* + (*gmnl l1*) + (*gmn x*))
∀ *x* ∈ *set l1*. (*is-measured-node x* ∧ *is-leveln-node n x*)
∀ *x* ∈ *set r1*. (*is-measured-node x* ∧ *is-leveln-node n x*)
*is-measured-node x* ∧ *is-leveln-node n x*
**by** *auto*
**from** *v2 v1 l1-x-r1-inv*(*4*) *pr-m-sf-inv* **have**
*ziel1*: *toList* (*Deep uu pr m sf*) = *toList l* @ *nodeToList nd* @ *toList r*
**by** (*auto simp add*: *nlistToTree-list deepL-list*)
**from** *l1-x-r1-inv*(*3*) *v1*(*1*) **have**
*v3*: *gmft l* = *gmnl l1* **by** (*simp add*: *gmnl-gmft*)
**with** *l1-x-r1-inv*(*1*,*2*) *v1* **have**
*ziel2*: ¬ *p* (*i* + *gmft l*)
*p* (*i* + *gmft l* + *gmn nd*)
**by** *simp-all*
**from** *l1-x-r1-inv*(*3*,*4*,*5*) *v1 v2*(*3*) *pr-m-sf-inv* **have**
*ziel3*: *is-leveln-ftree n l* ∧ *is-measured-ftree l* ∧
*is-leveln-ftree n r* ∧ *is-measured-ftree r* ∧
*is-leveln-node n nd* ∧ *is-measured-node nd*
**by** (*auto simp add*: *nlistToTree-inv deepL-inv*)
**from** *ziel1 ziel2 ziel3* **show** *?thesis* **by** *simp*

```
  qed
next
  case False note case1 = this with 3 show ?thesis
  proof (cases p (i + gmd pr + gmft m))
    case False with case1 3 show ?thesis
    proof −
      obtain l1 x r1 where
        l1xr1: splitDigit p (i + gmd pr + gmft m) sf = (l1,x,r1)
        by (cases splitDigit p (i + gmd pr + gmft m) sf, blast)
      with case1 False 3(7) have
        v1: l = deepR pr m l1 nd = x r = nlistToTree r1 by auto
      from l1xr1 have
        v2: digitToList sf = nlistToList l1 @ nodeToList x @ nlistToList r1
        length l1 ≤ 4 length r1 ≤ 4
        by (auto simp add: splitDigit-list)
      from 3(2,3) have
        pr-m-sf-inv: is-leveln-digit n pr ∧ is-measured-digit pr
        is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
        is-leveln-digit n sf ∧ is-measured-digit sf by simp-all
      from 3(3,6) have
        v7: p (i + gmd pr + gmft m + gmd sf) by (auto simp add: add.assoc)
      with pr-m-sf-inv 3(4) pr-m-sf-inv(3) case1 False l1xr1
          splitDigit-inv[of p i + gmd pr + gmft m sf n l1 x r1]
      have l1-x-r1-inv:
        ¬ p (i + gmd pr + gmft m + gmnl l1)
        p (i + gmd pr + gmft m + gmnl l1 + gmn x)
        ∀ x ∈ set l1. (is-measured-node x ∧ is-leveln-node n x)
        ∀ x ∈ set r1. (is-measured-node x ∧ is-leveln-node n x)
        is-measured-node x ∧ is-leveln-node n x
        by auto
      from v2 v1 l1-x-r1-inv(3) pr-m-sf-inv have
        ziel1: toList (Deep uu pr m sf) = toList l @ nodeToList nd @ toList r
        by (auto simp add: nlistToTree-list deepR-list)
      from l1-x-r1-inv(4) v1(3) have
        v3: gmft r = gmnl r1 by (simp add: gmnl-gmft)
      with l1-x-r1-inv(1,2,3) pr-m-sf-inv v1 v2 have
        ziel2: ¬ p (i + gmft l)
        p (i + gmft l + gmn nd)
        by (auto simp add: gmftR-gmnl add.assoc)
      from l1-x-r1-inv(3,4,5) v1 v2(2) pr-m-sf-inv have
        ziel3: is-leveln-ftree n l ∧ is-measured-ftree l ∧
        is-leveln-ftree n r ∧ is-measured-ftree r ∧
        is-leveln-node n nd ∧ is-measured-node nd
        by (auto simp add: nlistToTree-inv deepR-inv)
      from ziel1 ziel2 ziel3 show ?thesis by simp
    qed
  next
    case True with case1 3 show ?thesis
    proof −
```

**obtain** *l1 x r1* **where**
*l1-x-r1* :*nsplitTree p* (*i* + *gmd pr*) *m* = (*l1*, *x*, *r1*)
**by** (*cases nsplitTree p* (*i* + *gmd pr*) *m*) *blast*
**from** *3*(*2*,*3*) **have**
*pr-m-sf-inv*: *is-leveln-digit n pr* ∧ *is-measured-digit pr*
*is-leveln-ftree* (*Suc n*) *m* ∧ *is-measured-ftree m*
*is-leveln-digit n sf* ∧ *is-measured-digit sf* **by** *simp-all*
**with** *True case1*
*3.hyps*[*of i* + *gmd pr i* + *gmd pr* + *gmft m Suc n l1 x r1*]
*3*(*4*) *l1-x-r1*
**have** *l1-x-r1-inv*:
¬ *p* (*i* + *gmd pr* + *gmft l1*)
*p* (*i* + *gmd pr* + *gmft l1* + *gmn x*)
*is-leveln-ftree* (*Suc n*) *l1* ∧ *is-measured-ftree l1*
*is-leveln-ftree* (*Suc n*) *r1* ∧ *is-measured-ftree r1*
*is-leveln-node* (*Suc n*) *x* ∧ *is-measured-node x*
**and** *l1-x-r1-list*:
*toList m* = *toList l1* @ *nodeToList x* @ *toList r1*
**by** *auto*
**obtain** *l2 x2 r2* **where** *l2-x2-r2*:
*splitDigit p* (*i* + *gmd pr* + *gmft l1*) (*nodeToDigit x*) = (*l2*,*x2*,*r2*)
**by** (*cases splitDigit p* (*i* + *gmd pr* + *gmft l1*) (*nodeToDigit x*),*blast*)
**from** *l1-x-r1-inv*(*2*,*5*) **have**
*ndx-inv*: *is-leveln-digit n* (*nodeToDigit x*) ∧
*is-measured-digit* (*nodeToDigit x*)
*p* (*i* + *gmd pr* + *gmft l1* + *gmd* (*nodeToDigit x*))
**by** (*auto simp add*: *nodeToDigit-inv gmn-gmd*)
**note** *spdi* = *splitDigit-inv*[*of p i* + *gmd pr* + *gmft l1*
*nodeToDigit x n l2 x2 r2*]
**from** *ndx-inv l1-x-r1-inv*(*1*) *l2-x2-r2 3*(*4*) **have**
*l2-x2-r2-inv*:¬ *p* (*i* + *gmd pr* + *gmft l1* + *gmnl l2*)
*p* (*i* + *gmd pr* + *gmft l1* + *gmnl l2* + *gmn x2*)
∀ *x*∈*set l2*. *is-measured-node x* ∧ *is-leveln-node n x*
∀ *x*∈*set r2*. *is-measured-node x* ∧ *is-leveln-node n x*
*is-measured-node x2* ∧ *is-leveln-node n x2*
**by** (*auto simp add*: *spdi*)
**note** *spdl* = *splitDigit-list*[*of p i* + *gmd pr* + *gmft l1*
*nodeToDigit x l2 x2 r2*]
**from** *l2-x2-r2* **have**
*l2-x2-r2-list*:
*digitToList* (*nodeToDigit x*) =
*nlistToList l2* @ *nodeToList x2* @ *nlistToList r2*
*length l2* ≤ *4* ∧ *length r2* ≤ *4*
**by** (*auto simp add*: *spdl*)
**from** *case1 True 3*(*7*) *l1-x-r1 l2-x2-r2* **have**
*l-nd-r*:
*l* = *deepR pr l1 l2*
*nd* = *x2*
*r* = *deepL r2 r1 sf*

```
      by auto
    note dr1 = deepR-inv[OF l1-x-r1-inv(3) pr-m-sf-inv(1)]
    from dr1 l2-x2-r2-inv(3) l2-x2-r2-list(2) l-nd-r have
      l-inv: is-leveln-ftree n l ∧ is-measured-ftree l
      by simp
    note dl1 = deepL-inv[OF l1-x-r1-inv(4) pr-m-sf-inv(3)]
    from dl1 l2-x2-r2-inv(4) l2-x2-r2-list(2) l-nd-r have
      r-inv: is-leveln-ftree n r ∧ is-measured-ftree r
      by simp
    from l2-x2-r2-inv l-nd-r have
      nd-inv: is-leveln-node n nd ∧ is-measured-node nd
      by simp
    from l-nd-r(1,2) l2-x2-r2-inv(1,2,3)
        l1-x-r1-inv(3) l2-x2-r2-list(2) pr-m-sf-inv(1)
    have split-point:
      ¬ p (i + gmft l)
      p (i + gmft l + gmn nd)
      by (auto simp add: gmftR-gmnl add.assoc)
    from l2-x2-r2-list have x-list:
      nodeToList x = nlistToList l2 @ nodeToList x2 @ nlistToList r2
      by (simp add: nodeToDigit-list)
    from l1-x-r1-inv(3) pr-m-sf-inv(1)
        l2-x2-r2-inv(3) l2-x2-r2-list(2) l-nd-r(1)
    have l-list: toList l = digitToList pr @ toList l1 @ nlistToList l2
      by (auto simp add: deepR-list)
    from l1-x-r1-inv(4) pr-m-sf-inv(3) l2-x2-r2-inv(4)
        l2-x2-r2-list(2) l-nd-r(3)
    have r-list: toList r = nlistToList r2 @ toList r1 @ digitToList sf
      by (auto simp add: deepL-list)
    from x-list l1-x-r1-list l-list r-list l-nd-r
    have  toList (Deep uu pr m sf) = toList l @ nodeToList nd @ toList r
      by auto
    with split-point l-inv r-inv nd-inv
    show ?thesis by simp
  qed
 qed
qed
qed
```

A predicate on the elements of a monoid is called *monotone*, iff, when it holds for some value $a$, it also holds for all values $a + b$:

Split a finger tree by a monotone predicate on the annotations, using a given initial value. Intuitively, the elements are summed up from left to right, and the split is done when the predicate first holds for the sum. The predicate must not hold for the initial value of the summation, and must hold for the sum of all elements.

```
definition splitTree
  :: ('a::monoid-add ⇒ bool) ⇒ 'a ⇒ ('e, 'a) FingerTreeStruc
```

```
    ⇒ ('e, 'a) FingerTreeStruc × ('e × 'a) × ('e, 'a) FingerTreeStruc
  where
  splitTree p i t = (let (l, x, r) = nsplitTree p i t in (l, (n-unwrap x), r))

lemma splitTree-invpres:
  assumes inv: ft-invar (s:: ('e,'a::monoid-add) FingerTreeStruc)
  assumes init-ff: ¬ p i
  assumes sum-tt: p (i + annot s)
  assumes fmt: (splitTree p i s) = (l, (e,a), r)
  shows ft-invar l and ft-invar r
proof −
  obtain l1 nd r1 where
    l1-nd-r1: nsplitTree p i s = (l1, nd, r1)
    by (cases nsplitTree p i s, blast)

  with assms have
    l0: l = l1
    (e,a) = n-unwrap nd
    r = r1
    by (auto simp add: splitTree-def)
  note nsp = nsplitTree-invpres[of 0 s p i l1 nd r1]

  from assms have p (i + gmft s) by (simp add:  ft-invar-def annot-def)
  with assms l1-nd-r1 l0 have
    v1:
    is-leveln-ftree 0 l ∧ is-measured-ftree l
    is-leveln-ftree 0 r ∧ is-measured-ftree r
    is-leveln-node 0 nd  ∧ is-measured-node nd
    by (auto simp add: nsp ft-invar-def)
  thus ft-invar l and ft-invar r
    by (simp-all add: ft-invar-def annot-def)
qed


lemma splitTree-correct:
  assumes inv: ft-invar (s:: ('e,'a::monoid-add) FingerTreeStruc)
  assumes mono: ∀ a b. p a ⟶ p (a + b)
  assumes init-ff: ¬ p i
  assumes sum-tt: p (i + annot s)
  assumes fmt: (splitTree p i s) = (l, (e,a), r)
  shows (toList s) = (toList l) @ (e,a) # (toList r)
  and   ¬ p (i + annot l)
  and   p (i + annot l + a)
  and   ft-invar l and ft-invar r
proof −
  obtain l1 nd r1 where
    l1-nd-r1: nsplitTree p i s = (l1, nd, r1)
    by (cases nsplitTree p i s, blast)
```

```
  with assms have
    l0: l = l1
    (e,a) = n-unwrap nd
    r = r1
    by (auto simp add: splitTree-def)
  note nsp = nsplitTree-correct[of 0 s p i l1 nd r1]

  from assms have p (i + gmft s) by (simp add:  ft-invar-def annot-def)
  with assms l1-nd-r1 l0 have
    v1:
    (toList s) = (toList l) @ (nodeToList nd) @ (toList r)
    ¬ p (i + (gmft l))
    p (i + (gmft l) + (gmn nd))
    is-leveln-ftree 0 l ∧ is-measured-ftree l
    is-leveln-ftree 0 r ∧ is-measured-ftree r
    is-leveln-node 0 nd  ∧ is-measured-node nd
    by (auto simp add: nsp ft-invar-def)
  from v1(6) l0(2) have
    ndea: nd = Tip e a
    by (cases nd)  auto
  hence nd-list-inv: nodeToList nd = [(e,a)]
    gmn nd = a by simp-all
  with v1 show  (toList s) = (toList l) @ (e,a) # (toList r)
    and   ¬ p (i + annot l)
    and   p (i + annot l + a)
    and   ft-invar l and ft-invar r
    by (simp-all add: ft-invar-def annot-def)
qed

lemma splitTree-correctE:
  assumes inv: ft-invar (s:: ('e,'a::monoid-add) FingerTreeStruc)
  assumes mono: ∀ a b. p a ⟶ p (a + b)
  assumes init-ff: ¬ p i
  assumes sum-tt: p (i + annot s)
  obtains l e a r where
    (splitTree p i s) = (l, (e,a), r) and
    (toList s) = (toList l) @ (e,a) # (toList r) and
    ¬ p (i + annot l) and
    p (i + annot l + a) and
    ft-invar l and ft-invar r
proof −
  obtain l e a r where fmt: (splitTree p i s) = (l, (e,a), r)
    by (cases (splitTree p i s)) auto
  from splitTree-correct[of s p, OF assms fmt] fmt
  show ?thesis
    by (blast intro: that)
qed
```

### 1.2.8 Folding

**fun** *foldl-node* :: $('s \Rightarrow 'e \times 'a \Rightarrow 's) \Rightarrow 's \Rightarrow ('e,'a)\ Node \Rightarrow 's$ **where**
*foldl-node f σ (Tip e a) = f σ (e,a)|*
*foldl-node f σ (Node2 - a b) = foldl-node f (foldl-node f σ a) b|*
*foldl-node f σ (Node3 - a b c) =*
*foldl-node f (foldl-node f (foldl-node f σ a) b) c*

**primrec** *foldl-digit* :: $('s \Rightarrow 'e \times 'a \Rightarrow 's) \Rightarrow 's \Rightarrow ('e,'a)\ Digit \Rightarrow 's$ **where**
*foldl-digit f σ (One n1) = foldl-node f σ n1|*
*foldl-digit f σ (Two n1 n2) = foldl-node f (foldl-node f σ n1) n2|*
*foldl-digit f σ (Three n1 n2 n3) =*
*foldl-node f (foldl-node f (foldl-node f σ n1) n2) n3|*
*foldl-digit f σ (Four n1 n2 n3 n4) =*
*foldl-node f (foldl-node f (foldl-node f (foldl-node f σ n1) n2) n3) n4*

**primrec** *foldr-node* :: $('e \times 'a \Rightarrow 's \Rightarrow 's) \Rightarrow ('e,'a)\ Node \Rightarrow 's \Rightarrow 's$ **where**
*foldr-node f (Tip e a) σ = f (e,a) σ |*
*foldr-node f (Node2 - a b) σ = foldr-node f a (foldr-node f b σ)|*
*foldr-node f (Node3 - a b c) σ*
*= foldr-node f a (foldr-node f b (foldr-node f c σ))*

**primrec** *foldr-digit* :: $('e \times 'a \Rightarrow 's \Rightarrow 's) \Rightarrow ('e,'a)\ Digit \Rightarrow 's \Rightarrow 's$ **where**
*foldr-digit f (One n1) σ = foldr-node f n1 σ|*
*foldr-digit f (Two n1 n2) σ = foldr-node f n1 (foldr-node f n2 σ)|*
*foldr-digit f (Three n1 n2 n3) σ =*
*foldr-node f n1 (foldr-node f n2 (foldr-node f n3 σ))|*
*foldr-digit f (Four n1 n2 n3 n4) σ =*
*foldr-node f n1 (foldr-node f n2 (foldr-node f n3 (foldr-node f n4 σ)))*

**lemma** *foldl-node-correct*:
*foldl-node f σ nd = List.foldl f σ (nodeToList nd)*
**by** (*induct nd arbitrary*: *σ*) (*auto simp add*: *nodeToList-def*)

**lemma** *foldl-digit-correct*:
*foldl-digit f σ d = List.foldl f σ (digitToList d)*
**by** (*induct d arbitrary*: *σ*) (*auto*
*simp add*: *digitToList-def foldl-node-correct*)

**lemma** *foldr-node-correct*:
*foldr-node f nd σ = List.foldr f (nodeToList nd) σ*
**by** (*induct nd arbitrary*: *σ*) (*auto simp add*: *nodeToList-def*)

**lemma** *foldr-digit-correct*:
*foldr-digit f d σ = List.foldr f (digitToList d) σ*
**by** (*induct d arbitrary*: *σ*) (*auto*
*simp add*: *digitToList-def foldr-node-correct*)

Fold from left

**primrec** *foldl* :: $('s \Rightarrow 'e \times 'a \Rightarrow 's) \Rightarrow 's \Rightarrow ('e,'a)$ *FingerTreeStruc* $\Rightarrow 's$
**where**
*foldl f* $\sigma$ *Empty* $= \sigma$|
*foldl f* $\sigma$ (*Single nd*) = *foldl-node f* $\sigma$ *nd*|
*foldl f* $\sigma$ (*Deep - d1 m d2*) =
*foldl-digit f* (*foldl f* (*foldl-digit f* $\sigma$ *d1*) *m*) *d2*

**lemma** *foldl-correct*:
*foldl f* $\sigma$ *t* = *List.foldl f* $\sigma$ (*toList t*)
**by** (*induct t arbitrary*: $\sigma$) (*auto*
*simp add*: *toList-def foldl-node-correct foldl-digit-correct*)

Fold from right

**primrec** *foldr* :: $('e \times 'a \Rightarrow 's \Rightarrow 's) \Rightarrow ('e,'a)$ *FingerTreeStruc* $\Rightarrow 's \Rightarrow 's$
**where**
*foldr f Empty* $\sigma = \sigma$|
*foldr f* (*Single nd*) $\sigma$ = *foldr-node f nd* $\sigma$|
*foldr f* (*Deep - d1 m d2*) $\sigma$
= *foldr-digit f d1* (*foldr f m*(*foldr-digit f d2* $\sigma$))

**lemma** *foldr-correct*:
*foldr f t* $\sigma$ = *List.foldr f* (*toList t*) $\sigma$
**by** (*induct t arbitrary*: $\sigma$) (*auto*
*simp add*: *toList-def foldr-node-correct foldr-digit-correct*)

### 1.2.9 Number of elements

**primrec** *count-node* :: $('e, 'a)$ *Node* $\Rightarrow$ *nat* **where**
*count-node* (*Tip - a*) = *1* |
*count-node* (*Node2 - a b*) = *count-node a* + *count-node b* |
*count-node* (*Node3 - a b c*) = *count-node a* + *count-node b* + *count-node c*

**primrec** *count-digit* :: $('e,'a)$ *Digit* $\Rightarrow$ *nat* **where**
*count-digit* (*One a*) = *count-node a* |
*count-digit* (*Two a b*) = *count-node a* + *count-node b* |
*count-digit* (*Three a b c*) = *count-node a* + *count-node b* + *count-node c* |
*count-digit* (*Four a b c d*)
= *count-node a* + *count-node b* + *count-node c* + *count-node d*

**lemma** *count-node-correct*:
*count-node n* = *length* (*nodeToList n*)
**by** (*induct n*,*auto simp add*: *nodeToList-def count-node-def*)

**lemma** *count-digit-correct*:
*count-digit d* = *length* (*digitToList d*)
**by** (*cases d*, *auto simp add*: *digitToList-def count-digit-def count-node-correct*)

**primrec** *count* :: $('e,'a)$ *FingerTreeStruc* $\Rightarrow$ *nat* **where**
*count Empty* = *0* |

```
  count (Single a) = count-node a |
  count (Deep - pr m sf) = count-digit pr + count m + count-digit sf

lemma count-correct[simp]:
  count t = length (toList t)
  by (induct t,
    auto simp add: toList-def count-def
                   count-digit-correct count-node-correct)
end


interpretation FingerTreeStruc: FingerTreeStruc-loc .


no-notation FingerTreeStruc.lcons (infixr ◁ 65)
no-notation FingerTreeStruc.rcons (infixl ▷ 65)
```

## 1.3 Hiding the invariant

In this section, we define the datatype of all FingerTrees that fulfill their invariant, and define the operations to work on this datatype. The advantage is, that the correctness lemmas do no longer contain explicit invariant predicates, what makes them more handy to use.

### 1.3.1 Datatype

```
typedef (overloaded) ('e, 'a) FingerTree =
  {t :: ('e, 'a::monoid-add) FingerTreeStruc. FingerTreeStruc.ft-invar t}
proof −
  have Empty ∈ ?FingerTree by (simp)
  then show ?thesis ..
qed

lemma Rep-FingerTree-invar[simp]: FingerTreeStruc.ft-invar (Rep-FingerTree t)
  using Rep-FingerTree by simp

lemma [simp]:
  FingerTreeStruc.ft-invar t ⟹ Rep-FingerTree (Abs-FingerTree t) = t
  using Abs-FingerTree-inverse by simp

lemma [simp, code abstype]: Abs-FingerTree (Rep-FingerTree t) = t
  by (rule Rep-FingerTree-inverse)

typedef (overloaded) ('e,'a) viewres =
  { r:: (('e × 'a) × ('e,'a::monoid-add) FingerTreeStruc) option .
    case r of None ⇒ True | Some (a,t) ⇒ FingerTreeStruc.ft-invar t}
  apply (rule-tac x=None in exI)
  apply auto
  done
```

**lemma** [*simp*, *code abstype*]: *Abs-viewres* (*Rep-viewres x*) = *x*
**by** (*rule Rep-viewres-inverse*)

**lemma** *Abs-viewres-inverse-None*[*simp*]:
*Rep-viewres* (*Abs-viewres None*) = *None*
**by** (*simp add*: *Abs-viewres-inverse*)

**lemma** *Abs-viewres-inverse-Some*:
*FingerTreeStruc.ft-invar t* ⟹
*Rep-viewres* (*Abs-viewres* (*Some* (*a*,*t*))) = *Some* (*a*,*t*)
**by** (*auto simp add*: *Abs-viewres-inverse*)

**definition** [*code*]: *extract-viewres-isNone r* == *Rep-viewres r* = *None*
**definition** [*code*]: *extract-viewres-a r* ==
*case* (*Rep-viewres r*) *of Some* (*a*,*t*) ⇒ *a*
**definition** *extract-viewres-t r* ==
*case* (*Rep-viewres r*) *of None* ⇒ *Abs-FingerTree Empty*
| *Some* (*a*,*t*) ⇒ *Abs-FingerTree t*
**lemma** [*code abstract*]: *Rep-FingerTree* (*extract-viewres-t r*) =
(*case* (*Rep-viewres r*) *of None* ⇒ *Empty* | *Some* (*a*,*t*) ⇒ *t*)
**apply** (*cases r*)
**apply** (*auto split*: *option.split option.split-asm*
*simp add*: *extract-viewres-t-def Abs-viewres-inverse-Some*)
**done**

**definition** *extract-viewres r* ==
*if extract-viewres-isNone r then None*
*else Some* (*extract-viewres-a r*, *extract-viewres-t r*)

**typedef** (**overloaded**) ('*e*,'*a*) *splitres* =
{ ((*l*,*a*,*r*):: (('*e*,'*a*) *FingerTreeStruc* × ('*e* × '*a*) × ('*e*,'*a*::*monoid-add*) *FingerTreeStruc*))
| *l a r*.
*FingerTreeStruc.ft-invar l* ∧ *FingerTreeStruc.ft-invar r*}
**apply** (*rule-tac x*=(*Empty*,*undefined*,*Empty*) **in** *exI*)
**apply** *auto*
**done**

**lemma** [*simp*, *code abstype*]: *Abs-splitres* (*Rep-splitres x*) = *x*
**by** (*rule Rep-splitres-inverse*)

**lemma** *Abs-splitres-inverse*:
*FingerTreeStruc.ft-invar r* ⟹ *FingerTreeStruc.ft-invar s* ⟹
*Rep-splitres* (*Abs-splitres* ((*r*,*a*,*s*))) = (*r*,*a*,*s*)
**by** (*auto simp add*: *Abs-splitres-inverse*)

**definition** [*code*]: *extract-splitres-a r* == *case* (*Rep-splitres r*) *of* (*l*,*a*,*s*) ⇒ *a*
**definition** *extract-splitres-l r* == *case* (*Rep-splitres r*) *of* (*l*,*a*,*r*) ⇒

*Abs-FingerTree l*
**lemma** [*code abstract*]: *Rep-FingerTree* (*extract-splitres-l r*) = (*case* (*Rep-splitres r*) *of* (*l*,*a*,*r*) ⇒ *l*)
**apply** (*cases r*)
**apply** (*auto split*: *option.split option.split-asm simp add*: *extract-splitres-l-def Abs-splitres-inverse*)
**done**
**definition** *extract-splitres-r r* == *case* (*Rep-splitres r*) *of* (*l*,*a*,*r*) ⇒ *Abs-FingerTree r*
**lemma** [*code abstract*]: *Rep-FingerTree* (*extract-splitres-r r*) = (*case* (*Rep-splitres r*) *of* (*l*,*a*,*r*) ⇒ *r*)
**apply** (*cases r*)
**apply** (*auto split*: *option.split option.split-asm simp add*: *extract-splitres-r-def Abs-splitres-inverse*)
**done**

**definition** *extract-splitres r* ==
(*extract-splitres-l r*,
*extract-splitres-a r*,
*extract-splitres-r r*)

### 1.3.2 Definition of Operations

**locale** *FingerTree-loc*
**begin**
**definition** [*code*]: *toList t* == *FingerTreeStruc.toList* (*Rep-FingerTree t*)
**definition** *empty* **where** *empty* == *Abs-FingerTree FingerTreeStruc.Empty*
**lemma** [*code abstract*]: *Rep-FingerTree empty* = *FingerTreeStruc.Empty*
**by** (*simp add*: *empty-def*)

**lemma** *empty-rep*: *t*=*empty* ⟷ *Rep-FingerTree t* = *Empty*
**apply** (*auto simp add*: *empty-def*)
**apply** (*metis Rep-FingerTree-inverse*)
**done**

**definition** [*code*]: *annot t* == *FingerTreeStruc.annot* (*Rep-FingerTree t*)
**definition** *toTree t* == *Abs-FingerTree* (*FingerTreeStruc.toTree t*)
**lemma** [*code abstract*]: *Rep-FingerTree* (*toTree t*) = *FingerTreeStruc.toTree t*
**by** (*simp add*: *toTree-def*)
**definition** *lcons a t* ==
*Abs-FingerTree* (*FingerTreeStruc.lcons a* (*Rep-FingerTree t*))
**lemma** [*code abstract*]:
*Rep-FingerTree* (*lcons a t*) = (*FingerTreeStruc.lcons a* (*Rep-FingerTree t*))
**by** (*simp add*: *lcons-def FingerTreeStruc.lcons-correct*)
**definition** *rcons t a* ==
*Abs-FingerTree* (*FingerTreeStruc.rcons* (*Rep-FingerTree t*) *a*)
**lemma** [*code abstract*]:
*Rep-FingerTree* (*rcons t a*) = (*FingerTreeStruc.rcons* (*Rep-FingerTree t*) *a*)

```
  by (simp add: rcons-def FingerTreeStruc.rcons-correct)

definition viewL-aux t ==
  Abs-viewres (FingerTreeStruc.viewL (Rep-FingerTree t))
definition viewL t == extract-viewres (viewL-aux t)
lemma [code abstract]:
  Rep-viewres (viewL-aux t) = (FingerTreeStruc.viewL (Rep-FingerTree t))
  apply (cases (FingerTreeStruc.viewL (Rep-FingerTree t)))
  apply (auto simp add: viewL-aux-def )
  apply (cases Rep-FingerTree t = Empty)
  apply simp
  apply (auto
    elim!: FingerTreeStruc.viewL-correct-nonEmpty
              [of Rep-FingerTree t, simplified]
    simp add: Abs-viewres-inverse-Some)
  done

definition viewR-aux t ==
  Abs-viewres (FingerTreeStruc.viewR (Rep-FingerTree t))
definition viewR t == extract-viewres (viewR-aux t)
lemma [code abstract]:
  Rep-viewres (viewR-aux t) = (FingerTreeStruc.viewR (Rep-FingerTree t))
  apply (cases (FingerTreeStruc.viewR (Rep-FingerTree t)))
  apply (auto simp add: viewR-aux-def )
  apply (cases Rep-FingerTree t = Empty)
  apply simp
  apply (auto
    elim!: FingerTreeStruc.viewR-correct-nonEmpty
              [of Rep-FingerTree t, simplified]
    simp add: Abs-viewres-inverse-Some)
  done

definition [code]: isEmpty t == FingerTreeStruc.isEmpty (Rep-FingerTree t)
definition [code]: head t = FingerTreeStruc.head (Rep-FingerTree t)
definition tail t ≡
  if t=empty then
    empty
  else
    Abs-FingerTree (FingerTreeStruc.tail (Rep-FingerTree t))
  — Make function total, to allow abstraction
lemma [code abstract]: Rep-FingerTree (tail t) =
  (if (FingerTreeStruc.isEmpty (Rep-FingerTree t)) then Empty
   else FingerTreeStruc.tail (Rep-FingerTree t))
 apply (simp add: tail-def FingerTreeStruc.tail-correct FingerTreeStruc.isEmpty-def
empty-rep)
  apply (auto simp add: empty-def)
  done

definition [code]: headR t = FingerTreeStruc.headR (Rep-FingerTree t)
```

**definition** *tailR t* ≡
*if t=empty then*
*empty*
*else*
*Abs-FingerTree* (*FingerTreeStruc.tailR* (*Rep-FingerTree t*))
**lemma** [*code abstract*]: *Rep-FingerTree* (*tailR t*) =
(*if* (*FingerTreeStruc.isEmpty* (*Rep-FingerTree t*)) *then Empty*
*else FingerTreeStruc.tailR* (*Rep-FingerTree t*))
**apply** (*simp add*: *tailR-def FingerTreeStruc.tailR-correct FingerTreeStruc.isEmpty-def empty-rep*)
**apply** (*simp add*: *empty-def*)
**done**

**definition** *app s t* = *Abs-FingerTree* (
*FingerTreeStruc.app* (*Rep-FingerTree s*) (*Rep-FingerTree t*))
**lemma** [*code abstract*]:
*Rep-FingerTree* (*app s t*) =
*FingerTreeStruc.app* (*Rep-FingerTree s*) (*Rep-FingerTree t*)
**by** (*simp add*: *app-def FingerTreeStruc.app-correct*)

**definition** *splitTree-aux p i t* == *if* (¬*p i* ∧ *p* (*i*+*annot t*)) *then*
*Abs-splitres* (*FingerTreeStruc.splitTree p i* (*Rep-FingerTree t*))
*else*
*Abs-splitres* (*Empty*,*undefined*,*Empty*)
**definition** *splitTree p i t* == *extract-splitres* (*splitTree-aux p i t*)

**lemma** [*code abstract*]:
*Rep-splitres* (*splitTree-aux p i t*) = (*if* (¬*p i* ∧ *p* (*i*+*annot t*)) *then*
(*FingerTreeStruc.splitTree p i* (*Rep-FingerTree t*))
*else*
(*Empty*,*undefined*,*Empty*))
**using** *FingerTreeStruc.splitTree-invpres*[*of Rep-FingerTree t p i*]
**apply** (*auto simp add*: *splitTree-aux-def annot-def Abs-splitres-inverse*)
**apply** (*cases FingerTreeStruc.splitTree p i* (*Rep-FingerTree t*))
**apply** (*force simp add*: *Abs-FingerTree-inverse Abs-splitres-inverse*)
**done**

**definition** *foldl* **where**
[*code*]: *foldl f σ t* == *FingerTreeStruc.foldl f σ* (*Rep-FingerTree t*)
**definition** *foldr* **where**
[*code*]: *foldr f t σ* == *FingerTreeStruc.foldr f* (*Rep-FingerTree t*) *σ*
**definition** *count* **where**
[*code*]: *count t* == *FingerTreeStruc.count* (*Rep-FingerTree t*)

### 1.3.3 Correctness statements

**lemma** *empty-correct*: *toList t* = [] ⟷ *t*=*empty*
**apply** (*unfold toList-def empty-rep*)
**apply** (*simp add*: *FingerTreeStruc.toList-empty*)

**done**

**lemma** *toList-of-empty*[*simp*]: *toList empty* = []
**apply** (*unfold toList-def empty-def*)
**apply** (*auto simp add*: *FingerTreeStruc.toList-empty*)
**done**

**lemma** *annot-correct*: *annot t* = *sum-list* (*map snd* (*toList t*))
**apply** (*unfold toList-def annot-def*)
**apply** (*simp add*: *FingerTreeStruc.annot-correct*)
**done**

**lemma** *toTree-correct*: *toList* (*toTree l*) = *l*
**apply** (*unfold toList-def toTree-def*)
**apply** (*simp add*: *FingerTreeStruc.toTree-correct*)
**done**

**lemma** *lcons-correct*: *toList* (*lcons a t*) = *a*#*toList t*
**apply** (*unfold toList-def lcons-def*)
**apply** (*simp add*: *FingerTreeStruc.lcons-correct*)
**done**

**lemma** *rcons-correct*: *toList* (*rcons t a*) = *toList t*@[*a*]
**apply** (*unfold toList-def rcons-def*)
**apply** (*simp add*: *FingerTreeStruc.rcons-correct*)
**done**

**lemma** *viewL-correct*:
*t* = *empty* ⟹ *viewL t* = *None*
*t* ≠ *empty* ⟹ ∃ *a s*. *viewL t* = *Some* (*a*,*s*) ∧ *toList t* = *a*#*toList s*
**apply** (*unfold toList-def viewL-def viewL-aux-def*
*extract-viewres-def extract-viewres-isNone-def*
*extract-viewres-a-def*
*extract-viewres-t-def*
*empty-rep*)
**apply** (*simp add*: *FingerTreeStruc.viewL-correct*)
**apply** (*drule FingerTreeStruc.viewL-correct*(*2*)[*OF Rep-FingerTree-invar*])
**apply** (*auto simp add*: *Abs-viewres-inverse*)
**done**

**lemma** *viewL-empty*[*simp*]: *viewL empty* = *None*
**using** *viewL-correct* **by** *auto*

**lemma** *viewL-nonEmpty*:
**assumes** *t*≠*empty*
**obtains** *a s* **where** *viewL t* = *Some* (*a*,*s*) *toList t* = *a*#*toList s*
**using** *assms viewL-correct* **by** *blast*

**lemma** *viewR-correct*:

```
  t = empty ⟹ viewR t = None
  t ≠ empty ⟹ ∃ a s. viewR t = Some (a,s) ∧ toList t = toList s@[a]
  apply (unfold toList-def viewR-def viewR-aux-def
    extract-viewres-def extract-viewres-isNone-def
    extract-viewres-a-def
    extract-viewres-t-def
    empty-rep)
  apply (simp add: FingerTreeStruc.viewR-correct)
  apply (drule FingerTreeStruc.viewR-correct(2)[OF Rep-FingerTree-invar])
  apply (auto simp add: Abs-viewres-inverse)
  done

lemma viewR-empty[simp]: viewR empty = None
  using viewR-correct by auto

lemma viewR-nonEmpty:
  assumes t≠empty
  obtains a s where viewR t = Some (a,s) toList t = toList s@[a]
  using assms viewR-correct by blast

lemma isEmpty-correct: isEmpty t ⟷ t=empty
  apply (unfold toList-def isEmpty-def empty-rep)
 apply (simp add: FingerTreeStruc.isEmpty-correct FingerTreeStruc.toList-empty)
  done

lemma head-correct: t≠empty ⟹ head t = hd (toList t)
  apply (unfold toList-def head-def empty-rep)
  apply (simp add: FingerTreeStruc.head-correct)
  done

lemma tail-correct: t≠empty ⟹ toList (tail t) = tl (toList t)
  apply (unfold toList-def tail-def empty-rep)
  apply (simp add: FingerTreeStruc.tail-correct)
  done

lemma headR-correct: t≠empty ⟹ headR t = last (toList t)
  apply (unfold toList-def headR-def empty-rep)
  apply (simp add: FingerTreeStruc.headR-correct)
  done

lemma tailR-correct: t≠empty ⟹ toList (tailR t) = butlast (toList t)
  apply (unfold toList-def tailR-def empty-rep)
  apply (simp add: FingerTreeStruc.tailR-correct)
  done

lemma app-correct: toList (app s t) = toList s @ toList t
  apply (unfold toList-def app-def)
  apply (simp add: FingerTreeStruc.app-correct)
  done
```

**lemma** *splitTree-correct*:
**assumes** *mono*: $\forall\, a\ b.\ p\ a \longrightarrow p\ (a + b)$
**assumes** *init-ff*: $\neg\ p\ i$
**assumes** *sum-tt*: $p\ (i + annot\ s)$
**assumes** *fmt*: $(splitTree\ p\ i\ s) = (l,\ (e,a),\ r)$
**shows** $(toList\ s) = (toList\ l)\ @\ (e,a)\ \#\ (toList\ r)$
**and** $\neg\ p\ (i + annot\ l)$
**and** $p\ (i + annot\ l + a)$
**apply** (*rule*
*FingerTreeStruc.splitTree-correctE*[
**where** *p*=*p* **and** *s*=*Rep-FingerTree s*,
*OF - mono init-ff sum-tt*[*unfolded annot-def*],
*simplified*
])
**using** *fmt*
**apply** (*unfold toList-def splitTree-aux-def splitTree-def annot-def*
*extract-splitres-def extract-splitres-l-def*
*extract-splitres-a-def extract-splitres-r-def*) [*1*]
**apply** (*auto split*: *if-split-asm prod.split-asm*
*simp add*: *init-ff sum-tt*[*unfolded annot-def*] *Abs-splitres-inverse*) [*1*]

**apply** (*rule*
*FingerTreeStruc.splitTree-correctE*[
**where** *p*=*p* **and** *s*=*Rep-FingerTree s*,
*OF - mono init-ff sum-tt*[*unfolded annot-def*],
*simplified*
])
**using** *fmt*
**apply** (*unfold toList-def splitTree-aux-def splitTree-def annot-def*
*extract-splitres-def extract-splitres-l-def*
*extract-splitres-a-def extract-splitres-r-def*) [*1*]
**apply** (*auto split*: *if-split-asm prod.split-asm*
*simp add*: *init-ff sum-tt*[*unfolded annot-def*] *Abs-splitres-inverse*) [*1*]

**apply** (*rule*
*FingerTreeStruc.splitTree-correctE*[
**where** *p*=*p* **and** *s*=*Rep-FingerTree s*,
*OF - mono init-ff sum-tt*[*unfolded annot-def*],
*simplified*
])
**using** *fmt*
**apply** (*unfold toList-def splitTree-aux-def splitTree-def annot-def*
*extract-splitres-def extract-splitres-l-def*
*extract-splitres-a-def extract-splitres-r-def*) [*1*]
**apply** (*auto split*: *if-split-asm prod.split-asm*
*simp add*: *init-ff sum-tt*[*unfolded annot-def*] *Abs-splitres-inverse*) [*1*]
**done**

```
  lemma splitTree-correctE:
    assumes mono: ∀ a b. p a ⟶ p (a + b)
    assumes init-ff: ¬ p i
    assumes sum-tt: p (i + annot s)
    obtains l e a r where
    (splitTree p i s) = (l, (e,a), r) and
    (toList s) = (toList l) @ (e,a) # (toList r) and
    ¬ p (i + annot l) and
    p (i + annot l + a)
  proof −
    obtain l e a r where fmt: (splitTree p i s) = (l, (e,a), r)
      by (cases (splitTree p i s)) auto
    from splitTree-correct[of p, OF assms fmt] fmt
    show ?thesis
      by (blast intro: that)
  qed

  lemma foldl-correct: foldl f σ t = List.foldl f σ (toList t)
    apply (unfold toList-def foldl-def)
    apply (simp add: FingerTreeStruc.foldl-correct)
    done

  lemma foldr-correct: foldr f t σ = List.foldr f (toList t) σ
    apply (unfold toList-def foldr-def)
    apply (simp add: FingerTreeStruc.foldr-correct)
    done

  lemma count-correct: count t = length (toList t)
    apply (unfold toList-def count-def)
    apply (simp add: FingerTreeStruc.count-correct)
    done


end

interpretation FingerTree: FingerTree-loc .
```

## 1.4 Interface Documentation

In this section, we list all supported operations on finger trees, along with a short plaintext documentation and their correctness statements.

*FingerTree.toList*::$('a, 'b)$ *FingerTree* $\Rightarrow$ $('a \times 'b)$ *list*
Convert to list ($O(n)$)

*FingerTree.empty*::$('a, 'b)$ *FingerTree*
The empty finger tree ($O(1)$)
**Spec** *FingerTree.empty-correct*:

$(FingerTree.toList\ ?t = []) = (?t = FingerTree.empty)$

*FingerTree.annot*::$('a, 'b)$ *FingerTree* $\Rightarrow$ $'b$
Return sum of all annotations ($O(1)$)
**Spec** *FingerTree.annot-correct*:

$FingerTree.annot\ ?t = sum\text{-}list\ (map\ snd\ (FingerTree.toList\ ?t))$

*FingerTree.toTree*::$('a \times 'b)$ *list* $\Rightarrow$ $('a, 'b)$ *FingerTree*
Convert list to finger tree ($O(n \log(n))$)
**Spec** *FingerTree.toTree-correct*:

$FingerTree.toList\ (FingerTree.toTree\ ?l) = ?l$

*FingerTree.lcons*::$'a \times 'b \Rightarrow ('a, 'b)$ *FingerTree* $\Rightarrow$ $('a, 'b)$ *FingerTree*
Append element at the left end ($O(\log(n))$, $O(1)$ amortized)
**Spec** *FingerTree.lcons-correct*:

$FingerTree.toList\ (FingerTree.lcons\ ?a\ ?t) = ?a\ \#\ FingerTree.toList\ ?t$

*FingerTree.rcons*::$('a, 'b)$ *FingerTree* $\Rightarrow$ $'a \times 'b \Rightarrow ('a, 'b)$ *FingerTree*
Append element at the right end ($O(\log(n))$, $O(1)$ amortized)
**Spec** *FingerTree.rcons-correct*:

$FingerTree.toList\ (FingerTree.rcons\ ?t\ ?a) = FingerTree.toList\ ?t\ @\ [?a]$

*FingerTree.viewL*::$('a, 'b)$ *FingerTree* $\Rightarrow$ $(('a \times 'b) \times ('a, 'b)$ *FingerTree*$)$ *option*
Detach leftmost element ($O(\log(n))$, $O(1)$ amortized)
**Spec** *FingerTree.viewL-correct*:

$?t = FingerTree.empty \Longrightarrow FingerTree.viewL\ ?t = None$
$?t \neq FingerTree.empty \Longrightarrow$
$\exists\, a\ s.\ FingerTree.viewL\ ?t = Some\ (a, s) \wedge$
$\quad FingerTree.toList\ ?t = a\ \#\ FingerTree.toList\ s$

*FingerTree.viewR*::$('a, 'b)$ *FingerTree* $\Rightarrow$ $(('a \times 'b) \times ('a, 'b)$ *FingerTree*$)$ *option*
Detach rightmost element ($O(\log(n))$, $O(1)$ amortized)
**Spec** *FingerTree.viewR-correct*:

$?t = FingerTree.empty \Longrightarrow FingerTree.viewR\ ?t = None$
$?t \neq FingerTree.empty \Longrightarrow$
$\exists\, a\ s.\ FingerTree.viewR\ ?t = Some\ (a,\ s) \wedge$
$\quad FingerTree.toList\ ?t = FingerTree.toList\ s\ @\ [a]$

*FingerTree.isEmpty*::$('a,\ 'b)\ FingerTree \Rightarrow bool$
Check whether tree is empty ($O(1)$)
**Spec** *FingerTree.isEmpty-correct*:

$FingerTree.isEmpty\ ?t = (?t = FingerTree.empty)$

*FingerTree.head*::$('a,\ 'b)\ FingerTree \Rightarrow 'a \times 'b$
Get leftmost element of non-empty tree ($O(\log(n))$)
**Spec** *FingerTree.head-correct*:

$?t \neq FingerTree.empty \Longrightarrow FingerTree.head\ ?t = hd\ (FingerTree.toList\ ?t)$

*FingerTree.tail*::$('a,\ 'b)\ FingerTree \Rightarrow ('a,\ 'b)\ FingerTree$
Get all but leftmost element of non-empty tree ($O(\log(n))$)
**Spec** *FingerTree.tail-correct*:

$?t \neq FingerTree.empty \Longrightarrow$
$FingerTree.toList\ (FingerTree.tail\ ?t) = tl\ (FingerTree.toList\ ?t)$

*FingerTree.headR*::$('a,\ 'b)\ FingerTree \Rightarrow 'a \times 'b$
Get rightmost element of non-empty tree ($O(\log(n))$)
**Spec** *FingerTree.headR-correct*:

$?t \neq FingerTree.empty \Longrightarrow FingerTree.headR\ ?t = last\ (FingerTree.toList\ ?t)$

*FingerTree.tailR*::$('a,\ 'b)\ FingerTree \Rightarrow ('a,\ 'b)\ FingerTree$
Get all but rightmost element of non-empty tree ($O(\log(n))$)
**Spec** *FingerTree.tailR-correct*:

$?t \neq FingerTree.empty \Longrightarrow$
$FingerTree.toList\ (FingerTree.tailR\ ?t) = butlast\ (FingerTree.toList\ ?t)$

*FingerTree.app*::$('a,\ 'b)\ FingerTree \Rightarrow ('a,\ 'b)\ FingerTree \Rightarrow ('a,\ 'b)\ FingerTree$
Concatenate two finger trees ($O(\log(m+n))$)
**Spec** *FingerTree.app-correct*:

$FingerTree.toList\ (FingerTree.app\ ?s\ ?t) =$
$FingerTree.toList\ ?s\ @\ FingerTree.toList\ ?t$

*FingerTree.splitTree*

*FingerTree.splitTree*::$('a \Rightarrow bool)$
$\Rightarrow 'a \Rightarrow ('b,\ 'a)\ FingerTree$
$\Rightarrow ('b,\ 'a)\ FingerTree \times$
$('b \times 'a) \times ('b,\ 'a)\ FingerTree$

Split tree by a monotone predicate. ($O(\log(n))$)

A predicate $p$ over the annotations is called monotone, iff, for all annotations $a, b$ with $p(a)$, we have already $p(a + b)$.

Splitting is done by specifying a monotone predicate $p$ that does not hold for the initial value $i$ of the summation, but holds for $i$ plus the sum of all annotations. The tree is then split at the position where $p$ starts to hold for the sum of all elements up to that position.

**Spec** *FingerTree.splitTree-correct*:

$$\llbracket \forall\, a\ b.\ ?p\ a \longrightarrow ?p\ (a + b);\ \neg\ ?p\ ?i;\ ?p\ (?i + \mathit{FingerTree.annot}\ ?s);\ \mathit{FingerTree.splitTree}\ ?p\ ?i\ ?s = (?l,\ (?e,\ ?a),\ ?r) \rrbracket \Longrightarrow \mathit{FingerTree.toList}\ ?s = \mathit{FingerTree.toList}\ ?l\ @\ (?e,\ ?a)\ \#\ \mathit{FingerTree.toList}\ ?r$$

$$\llbracket \forall\, a\ b.\ ?p\ a \longrightarrow ?p\ (a + b);\ \neg\ ?p\ ?i;\ ?p\ (?i + \mathit{FingerTree.annot}\ ?s);\ \mathit{FingerTree.splitTree}\ ?p\ ?i\ ?s = (?l,\ (?e,\ ?a),\ ?r) \rrbracket \Longrightarrow \neg\ ?p\ (?i + \mathit{FingerTree.annot}\ ?l)$$

$$\llbracket \forall\, a\ b.\ ?p\ a \longrightarrow ?p\ (a + b);\ \neg\ ?p\ ?i;\ ?p\ (?i + \mathit{FingerTree.annot}\ ?s);\ \mathit{FingerTree.splitTree}\ ?p\ ?i\ ?s = (?l,\ (?e,\ ?a),\ ?r) \rrbracket \Longrightarrow ?p\ (?i + \mathit{FingerTree.annot}\ ?l + ?a)$$

*FingerTree.foldl*

$$\mathit{FingerTree.foldl} :: ('a \Rightarrow 'b \times 'c \Rightarrow 'a) \Rightarrow 'a \Rightarrow ('b,\ 'c)\ \mathit{FingerTree} \Rightarrow 'a$$

Fold with function from left

**Spec** *FingerTree.foldl-correct*:

$$\mathit{FingerTree.foldl}\ ?f\ ?\sigma\ ?t = \mathit{foldl}\ ?f\ ?\sigma\ (\mathit{FingerTree.toList}\ ?t)$$

*FingerTree.foldr*

$$\mathit{FingerTree.foldr} :: ('a \times 'b \Rightarrow 'c \Rightarrow 'c) \Rightarrow ('a,\ 'b)\ \mathit{FingerTree} \Rightarrow 'c \Rightarrow 'c$$

Fold with function from right

**Spec** *FingerTree.foldr-correct*:

$$\mathit{FingerTree.foldr}\ ?f\ ?t\ ?\sigma = \mathit{foldr}\ ?f\ (\mathit{FingerTree.toList}\ ?t)\ ?\sigma$$

*FingerTree.count*$::('a,\ 'b)\ \mathit{FingerTree} \Rightarrow \mathit{nat}$

Return the number of elements

**Spec** *FingerTree.count-correct*:

$$\mathit{FingerTree.count}\ ?t = \mathit{length}\ (\mathit{FingerTree.toList}\ ?t)$$

**end**

## 2 Related work

Finger trees were originally introduced by Hinze and Paterson[1], who give an implementation in Haskell. Our implementation closely follows this original implementation.

There is also a machine-checked formalization of 2-3 finger trees in Coq [2]. Like ours, it closely follows the original paper of Hinze and Paterson. The main difference is that the Coq-formalization encodes the invariants directly into the datatype for finger trees, while we first define the bigger algebraic datatype *FingerTreeStruc* along with the predicate *ft-invar* that checks the invariant. This bigger type and the *ft-invar*-predicate is then wrapped into the datatype *FingerTree*, that, however, exposes no algebraic structure any more. Our approach greatly simplifies matters in the context of Isabelle/HOL, as it can be realized with Isabelle's datatype-package.

## References


[1] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.

[2] M. Sozeau. Program-ing finger trees in coq. In *ICFP '07*, pages 13–24, New York, NY, USA, 2007. ACM.